Various Algorithms for High Throughput Sequencing

by

Vladimir Yanovsky

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Various Algorithms for High Throughput Sequencing

Vladimir Yanovsky
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2014

The growing volume of generated DNA sequencing data makes the problem of its long term storage increasingly important. In the first chapter we present ReCoil – an I/O efficient external memory algorithm designed for compression of very large datasets of short reads DNA data. Typically each position of DNA sequence is covered by multiple reads of a short read dataset and our algorithm makes use of resulting redundancy to achieve high compression rate. While compression based on encoding mismatches between the dataset and a similar reference can yield high compression rate, good quality reference sequence may be unavailable. Instead, ReCoil's compression is based on encoding the differences between similar or overlapping reads. As such reads may appear at large distances from each other in the dataset and since random access memory is a limited resource, ReCoil is designed to work efficiently in external memory, leveraging high bandwidth of modern hard disk drives.

The next problem we address is the problem of mapping sequencing data to the reference genome. First, we show an algorithm for aligning two-pass Single Molecule Sequencing reads. Single Molecule Sequencing technologies such as the Heliscope simplify the preparation of DNA for sequencing, while sampling millions of reads in a day. The technology suffers from a high error rate, ameliorated by the ability to sample multiple reads from the same location. We develop novel rapid alignment algorithms for two-pass Single Molecule Sequencing methods. We combine the Weighted Sequence Graph representation of all optimal and near optimal alignments between the two reads sampled from a piece of DNA with k-mer filtering methods and spaced seeds to quickly generate candidate locations for the reads on the reference genome. Our method combines these approaches with fast implementation of the Smith-Waterman algorithm in order to build an algorithm that is both fast and accurate, since it is able to take complete advantage of both of the reads sampled during two pass sequencing.

Next we revisit the basic problem of mapping a read to the reference. We develop a fast, memory-efficient algorithm for mapping of the new generation of long high error

rate reads reads. We introduce a novel approach to hash based indexing which is both succinct and can be constructed very efficiently, avoiding the need to build the index in advance. Using our new index, we design a cache oblivious algorithm that makes an efficient use of the modern hardware to quickly cluster the reads according to their position on the reference.

While genetic sequencing is growing in popularity as the tool to answer various kinds of questions about the genome, technical as well as practical considerations, such as high cost, impose limitations on its use. Microarrays are a commonly used alternative to sequencing for the task of microsatellite genotyping. We discuss the problem of optimal coloring of dotted interval graphs that arises naturally in the context of microsatelite genotyping using microarrays.

Relation to published work

Chapter 2 was partially published as:

- "ReCoil – an algorithm for compression of extremely large datasets of dna data.", Vladimir Yanovsky, Algorithms for Molecular Biology, 6:23, 2011, [71].

Chapter 3 was partially published as:

- "Read mapping algorithms for single molecule sequencing data." Vladimir Yanovsky, Stephen M. Rumble, and Michael Brudno. In Keith A. Crandall and Jens Lagergren, editors, WABI, volume 5251 of Lecture Notes in Computer Science, pages 38-49, Springer, 2008, [72]

Chapter 5 was partially published as:

- "Approximation algorithm for coloring of dotted interval graphs." Vladimir Yanovsky. Inf. Process. Lett., 108(1):41-44, September 2008, [70].

Contents

Chapter 1

Introduction

1.1 Sequencing

Genome sequencing is a process used to determine DNA sequence of an organism. As a typical DNA strand is longer than current technology allows to sequence in one step, a method, called *shotgun* sequencing, is used for longer DNA sequences. With this method a DNA strand is first replicated several times, producing several copies of the original sequence. Next each of the copies is sheared randomly into shorter sequences, called *reads*. Different copies of the sequence are split in different places allowing to use the overlaps between reads in order to infer information about their locations on the original sequence.

While classical, Sanger-style sequencing machines were able to sequence 500 thousand basepairs per run at a cost of over $1000 per megabase, new High Throughput Sequencing (HTS) technologies, such as Solexa/Illumina and AB SOLiD can sequence billions of nucleotides in the same amount of time, at the cost of only $1 per megabase. The decreased cost and higher throughput of the new technologies, however, is offset with higher error rates. One disadvantage of earlier generations of HTS was their relatively short read length – dozens of nucleotides – though with advance of HTS methods it became possible to generate reads hundreds of nucleotides long.

Most sequencing technologies reduce the cost of sequencing by running many sequencing experiments in parallel. After the input DNA is sheared into smaller pieces, DNA libraries are prepared by a Polymerase Chain Reaction (PCR) method, with many identical copies of the molecules created, and attached to a particular location on a slide. All of the molecules are sequenced in parallel using sequencing by hybridization or ligation, with each nucleotide producing a specific color as it is sequenced. The colors for all of the positions on the slide are recorded via imaging and are then converted into base calls.

More recently another type of sequencing technology, called *Single Molecule Sequencing (SMS)*, has started to enter into the mainstream. While the origins of SMS date back to 1989 [20], it is only now becoming practical. The *Heliscope Genetic Analysis System* [21] developed by *Helicos* was the first commercial product that allows for the sequencing of DNA with SMS.

The key advantage of the SMS methods over other HTS technologies is the direct se-

quencing of DNA, without the need for the PCR step of the more established technologies we described above. PCR has different selection rates for different DNA molecules, and introduces substitutions into the DNA sequence as it is copied. SMS methods should also lead to more accurate estimates of the quantity of DNA which is required for detection of copy number variations and quantification of gene expression levels via sequencing. SMS technologies have very different error distribution than classical Sanger-style sequencing. Because only one physical piece of DNA is sequenced at a time, the sequencing signal is much weaker, leading to a large number of "dark bases": nucleotides that are skipped during the sequencing process leading to deletion errors. Additionally, a nucleotide could be misread (substitution errors), or inserted, however these errors are more typically the result of imaging, rather than chemistry problems, and hence are rarer. For the case of the Heliscope sequencer, a full description of the errors is in the supplementary material of [21].

An important advantage of the Heliscope sequencer, as well as several other proposed SMS technologies (e.g. Pacific Biosciences') is the ability to read a particular piece of DNA multiple times. This property is called multi-pass sequencing. The availability of multiple reads from a single piece of DNA can help confirm the base calls, and reduce the overall error rate. In practice these methods usually use two passes, as this offers a good tradeoff between error rate and cost.

1.2 External Memory Compression of Sequencing Data

1.2.1 Problem Overview

High speeds and relatively low prices of HTS technologies led to their widespread use for various kinds of applications, making it important to store high volumes of generated sequencing data efficiently. HTS reads frequently have relatively high error rates. On the other hand, the datasets produced by them are usually of high *coverage*, i.e. they can have many different reads overlapping at each position, making them highly compressible.

1.2.2 DNA Sequence Compression

DNA sequence contains a large number of approximate repeats. Yet, general purpose compression tools, such as *gzip* or *bzip2*, cannot make use of this redundancy in order to achieve compression rate for DNA sequences or datasets significantly better than the trivial encoding of two bits for each of four possible nucleotides [12].

Specialized DNA compression algorithms find approximate repeats in the sequence and then attempt to encode efficiently the differences between the instances of the repeat. The best compression to date for a single sequence is achieved by DNACompress [12]. This tool is based on PatternHunter [45] – a package for sequence homology search similar to BLAST. DNACompress runs PatternHunter to find approximate repeats in the sequence, then sorts them such that long high similarity repeats appear first. During the encoding stage DNACompress extracts the highest scoring approximate repeat and encodes all its instances using edit operations transforming between them. Then the list

of all hits reported by PatternHunter is filtered out of all sequences overlapping with those encoded by the step. This step is repeated until the remaining highest scoring hit has the score below some threshold. While it is possible to modify DNACompress for the compression of the datasets of HTS reads, it is not designed to handle large input size: in [67] the authors tested DNACompress and found it could not handle even the smallest of their datasets.

Genomic Dataset Compressed Indexing

Several works consider the problem of compressed full-text self-indexing of a DNA dataset. For example, Makinen et al. [46] describes an algorithm to build a compressed *self-index* for a collection of strings of equal length generated by the SNPs in a single string. The index they introduce is a suffix array based data structure that given a string T permits the following operations:

- $Count(T)$ – counts the number of appearances of T as a substring of the strings in the collection
- $Search(T)$ – outputs the positions where T appears as a substring in the strings of the collection
- $Display(i, j, k)$ – displays $S_k[i \cdots j]$, where S_k is the k'th string of the collection

While compressed full text indices achieve lower compression rate than known dedicated compression utilities [23], they address a different set of tradeoffs than our work, in which we attempt to achieve the best compression rate.

Compression Using a Known Reference Sequence

Knowing the reference genome makes it possible to achieve very high compression rate by storing only the differences between the sequences. In [14] this approach was used for compression of a single human genome. To further improve the compression rate, the algorithm stores the mutations in the target genome using public databases of known genomic variations. The result was just a 4 MB compressed file.

Other tools, such as SlimGene [39] and MZip [35], address the problem of compressing a dataset by mapping each read against a highly similar reference sequence and storing only the differences between the read and the reference. These tools receive as an input alignments of each read produced by a mapping program and use methods for variable length integer coding, such as Huffman coding, to efficiently store the alignments. In addition to compressing DNA sequences, SlimGene [39] and MZip [35] discuss ways to efficiently store the quality scores – the values attached to each position in the read, reflecting the confidence level of the sequencer in its output for that position. Due to relatively high space requirement of the quality scores, the works suggest to store them using lossy compression, arguing that, in practice, storing exact values would not be beneficial in most applications. In this work we only address compression of DNA sequences.

There are limitations to the reference-based approach. The reference sequence might not be available, for example for a metagenomic dataset, or not be similar enough for organisms with very high polymorphism. Also, there is a strong practical advantage of compression methods that keep all the data required for the decompression in the compressed file. Decoding a reference-based encoded sequence, especially following a transfer or long term storage, can be inconvenient, as the decoder must be able to access the reference, likely to be stored separately from the compressed dataset due to its size.

Reconstruction of the original (or assembled) sequence, if it is not given, and using it as the reference for storing the dataset is not a viable option: the problem of genome assembly, especially in the presence of sequencing errors, is computationally too expensive and there are no known *De Novo* assembly algorithms that would work on a workstation [75]. Hence we are interested in the problem of compression of a dataset under assumption that the original sequence is unknown.

Coil

White et al. [67] introduce *Coil* – a tool designed for compression of datasets of DNA sequences not relying on a known reference sequence. Coil builds a hash table H of the lists of locations of all DNA strings of length k, called k-mers, and uses it to find the subset of sequences similar to each sequence in the input dataset S. This subset can be found as follows: for a sequence S_i compute the set of all k-mers in S_i and use hash table H to merge the lists of sequences containing them. Those sequences that appear frequently enough in the merged list are likely to be similar to S_i. Since this merged list has to be found for each input sequence, in order to speed up execution, Coil uses the following Least Recently Used (LRU) heuristic for k-mers' list merging. When Coil computes the set of sequences similar to S_i, instead of merging all location lists for all k-mers in S_i into one possibly very long list, it manages only a fixed number r of small, fixed-size LRU arrays A_j, $0 \leq j < r$, which together hold the most recently accessed sequences that share a k-mer with S_i.

The sequences in each A_j are maintained in the order of time since their last sighting and for each sequence Coil counts the number of times the sequence was accessed since it was placed in A_j last time. Scanning through the lists of sequences containing each k-mer in S_i, each such sequence s is merged into $A_{s \bmod r}$ as follows: if s is already present, its counter is incremented and the sequence is moved to the front of the array; otherwise all sequences in the array are shifted back to make space for s, discarding the least recently seen sequence if the array was already full. The size of this array is necessarily small, otherwise the updates to it would be too expensive. This may result in sequences being placed in and removed from some A_j repeatedly in the cases when k-mer lists are expected to be long such as for smaller values of k and for datasets containing many short reads, like those produced by HTS.

In the next stage Coil builds a weighted *similarity graph* with the vertices corresponding to the sequences of the dataset and an edge between a sequence S_i and each sequence in each array A_j. The weight of each edge equals the corresponding appearance counter.

In the encoding step, in a manner reminiscent of phylogenetic tree approximations, Coil uses the Maximum Spanning Tree in the similarity graph as its encoding tree. Coil

stores the read in the root of the spanning tree explicitly and then encodes, in the order of a preorder traversal of the tree, each child with the set of differences between it and its parent. If the weight of the edit script is above some threshold, the child is stored explicitly. Finally, Coil compresses the scripts with a general purpose compression utility such as *bzip2*.

1.3 Mapping Reads to a Reference Sequence

In this section we give a general introduction to the problem of read mapping. An overview of the data structures relevant to the problem is deferred to Chapter 4.

1.3.1 Read Mapping Problem

The first step in using a set of DNA reads is usually mapping them, i.e. finding the locations of all reads on a known reference genome. The mapping information can be later used for a wide variety of tasks, such as checking on specific genes of the individual associated with the data set, finding variations between genomes or to assist in reconstructing of the complete genome sequence from the data set. Next we introduce the problem of read mapping, alternatively called read matching or alignment.

In the most general terms the read mapping problem can be seen as an approximate string matching: find locations on the reference string that are similar to the pattern string under some measure of similarity. Read mapping is a fundamental problem in computational biology and is used as a basic building block in many algorithms.

More precisely, given two strings: S, the reference, and r, a read, the goal is to find a set of substrings r' of S that minimize a distance measure $d(r; r')$. The measure of similarity, or the distance $d(r; r')$, is defined in a domain specific way. In the simplest case $d(r; r')$ is the sum of costs of edit transformations: substitutions, insertions or deletions, required to transform one string into another. The constant costs for each type of the edit transformations is defined using the available domain data, for example using known mutation rates in case DNA strings. If the cost of an edit transformation is one, independently of its type, we get a measure called Manhattan Edit Distance. This measure is frequently used to give an approximation on the domain-specific cost function, while being computationally more convenient.

1.3.2 Exact Solution - Smith-Waterman Algorithm

The classical Smith and Waterman [62] solution uses a dynamic programming (DP) approach to the problem of approximate string matching. Time complexity of their algorithm is $|r||S|$, making it too slow for practical applications. The questions on upper or lower bounds of this problem remain open.

Longest Common Subsequence (LCS) is a restricted case of the matching problem where insertion and deletion costs are zero. For LCS it was shown [2] that in comparison model upper and lower bounds are both quadratic. Yet in a more general model allowing O(1) arithmetic operations an $O(|r||S|/\min(\log|r|, \log|S|)$ algorithm for the problem

was proposed by [15]. No polynomially sub-quadratic algorithms for the problem are known.

1.3.3 Heuristic Mapping Algorithms

A mapping algorithm is typically required to align a multi-billion nucleotide dataset versus a multi-billion nucleotide reference in a matter of several hours, making exact algorithms not practical on current hardware. As result, all mapping algorithms use some heuristics to speed up the search and produce suboptimal alignments in some cases. A *filtering* algorithm allows to prune regions on the genome unlikely to contain a match. A filter can be either exact [56], when it is guaranteed that the best match is always found, or heuristic [59], when a small chance of reporting a suboptimal match is allowed.

The design goal of a heuristic algorithm is to minimize the number of errors it makes. Highly repetitive sequences are those that most often will be mapped suboptimally. Fortunately high copy count repeats do not usually carry significant biological information.

In order to match large sets of reads against a very long reference efficiently, most mapping algorithms *index* the reference, i.e. preprocess the reference in some way that facilitates pattern matching in it. While there are many differences between the mappers and their indexing approaches, all of them fall into one of two general indexing frameworks. The first is based on Burrows-Wheeler Transform(BWT) [9] and its duality with suffix arrays [47] and suffix trees. The second approach is a seed-based filtering.

1.3.4 Suffix Tree-Based Mapping

The index data structure behind most current mapping algorithms is a suffix tree or one of its memory efficient representations.

Suffix tree is a fundamental data structure that allows $O(r)$ exact time search for read r in reference S. $O(|S| \log |S|)$ memory usage of a pointer-based suffix tree representation is too high to be practical on most current hardware for strings of size of the human genome. Suffix array [47] is a representation of a suffix tree in an array. While more space efficient then suffix trees in practice, suffix arrays still have the same $O(|S| \log |S|)$ worst case complexity. The discoveries of duality between the suffix arrays and BWT transform by [30] and [25] allowed implementations of suffix array in $O(S)$ space or even less, in the case that S is compressible [29].

Suffix tree or suffix array based mapping is employed by several algorithms, such as Bowtie [40], BWA [43] and SSAHA [53] and is currently the fastest way to align high quality short reads. Usually these methods emulate in some way the search in a suffix tree and share the same limitation: handling errors and variation requires backtracking, the backtracking incurs high costs for low quality or long reads.

While there exist linear algorithms to construct both suffix arrays and suffix trees, the construction is relatively computationally intensive, usually taking hours of runtime to construct the index. Menon et al. [48] introduce a MapReduce based algorithm to construct a suffix array for full human genome in 9 minutes using 120 cores of the Amazon Elastic Compute Cloud (EC2), achieving 7.85x speedup, compared to a single core. Due

to significant cost to construct an index, compared to mapping runtime, all mapping programs require that the index is prepared in advance.

1.3.5 Seed-Based Mapping

Given a string S and an integer k, the set of substrings S of length k is called the set of k-mers in S. A generalization of a k-mer is a *spaced seed* [45] of weight k: a bit vector p of length l with k ones and $l - k$ zeros. The spaced seed at position i of S is a subsequence of length k: $S[i + j]|p[j] = 1$, i.e. bit vector p is used as a mask starting at $S[i]$, skipping characters of S corresponding to zeros in p. A k-mer is a special case of a spaced seed for $l = k$. In many cases spaced seeds have better sensitivity/selectivity ratio than contiguous seeds.

Given the maximum allowed number of errors in the match, one can use "pigeonhole principle" to obtain the lower bound on the number of k-mers appearing in the match. If a read r matches in a *candidate window* $w = S[i; i + h]$, where $h \approx |r|$ is an integer parameter, called a window length, then for sufficiently small value of k, the sets of k-mers of the read and of the candidate window must intersect.

Rasmussen et al. [56] used this idea in order to construct an exact filtering algorithm based on k-mers.

Selecting the value of k presents a tradeoff. If k is too small, the *selectivity* would be low as the k-mers will be likely to occur at many places of S just by chance. This becomes particularly problematic for short k-mers appearing in highly repetitive regions. On the other hand, if k is too large, the *sensitivity* is low. This makes the final dynamic programming verification step too expensive due to high number of candidate windows. Their flexibility and tolerance to errors make seed-based mapping the preferred candidate where high similarity between the reads and the reference cannot be guaranteed while high sensitivity is required [1].

There are two types of seed-based aligners: those that index the seeds in the reads and those that index the reference. SHRiMP [59] is an example of an aligner hashing reads. Most other seed-based approaches, such as SNAP [73], BFAST [34] as well as SHRiMP2 [16] hash the seeds in the reference.

SeqAlto [49] uses a memory-efficient implementation of an index of locations on the reference of relatively long seeds. SeqAlto targets datasets of high-quality reads longer than 100bp with low number of gaps as errors. The authors show that for reads satisfying these criteria the approach is accurate and fast.

Pacific Biosciences developed a tool, called BLASR, for the purpose of mapping long and high error rate reads. BLASR employs a BWT-based index to find the initial set of candidate alignments between the read and the reference which are subsequently chained, using interval chaining [50]. Interval chaining requires $\Omega(n \log n)$ time where n is the number of candidates. Finally, the results are verified using the dynamic programming. The problem with this approach is that it is very sensitive to the length of the initial candidate BWT-matches. The candidates must be long enough, otherwise the set of them would be too large and the chaining too expensive. On the other hand, due to high error rate the initial matches cannot be required to be too long.

Maximal Exact Match (MEM) is an exact match that cannot be extended in either direction of the match. Super Maximal Exact Match [41] is a MEM that is not contained in other MEMs between the read and the query sequence. BWA-MEM released recently [42] implements an algorithm that finds SMEMs between the read and the reference using a BWT-based algorithm. The positions on the read and the reference are chained next. The algorithm is able to backtrack on the length of SMEM if it is too long.

1.3.6 Vectorized Smith-Waterman Algorithm

Most contemporary mobile, desktop and server-class processors have special vector execution units, which perform multiple simultaneous data operations in a single instruction. For example, it is possible to add the eight individual 16-bit elements of two 128-bit vectors in one machine instruction. Over the past decade, several methods have been devised to significantly enhance the execution speed of Smith-Waterman-type algorithms by parallelizing the computation of several cells of the dynamic programming matrix. The simplest such implementation by Wozniak [69] computes the dynamic programming matrix using diagonals (See Figure 3.2B). Since each cell of the matrix can be computed once the cell immediately above, immediately to the left, and at the upper-left corner have been computed, one can compute each successive diagonal once the two prior diagonals have been completed. The problem can then be parallelized across the length of supported diagonals, often leading to a vectorization factor of 4 to 16. The only portion of such an approach that cannot be parallelized is the identification of match/mismatch scores for every cell of the matrix. These operations are done sequentially, necessitating 24 independent, expensive data loads for 8-cell vectors. The approach becomes increasingly problematic as vector sizes increase because memory loads cannot be vectorized; when the parallelism grows, so does the number of lookups.

Rognes and Seeberg [57] developed an alternative algorithm with the following innovations: first, they avoided the aforementioned loads by pre-calculating a 'query profile' before computing the matrix. By pre-allocating vectors with the appropriate scores in memory, they needed only load a single score vector on each computation of a vector of cells. The up-front cost in pre-calculation greatly reduced the expense of computing scores in the crucial, tight inner-loop of the algorithm. The query profile, however, requires that scores be computed per-column, rather than for each diagonal. This creates data dependencies between several cells within the vector. However, the authors took advantage of the fact that often cells contain the value 0 when computing the Smith-Waterman recurrence, and any gap continuation would therefore be provably suboptimal. This enables the conditional skipping of a significant number of steps. Although highly successful for protein sequences, where only 1/20 elements on average have a positive score, it is significantly less beneficial for DNA sequences where usually at least 1/4 elements in the matrix match. Farrar [22] most recently introduced a superior method: by striping the query profile layout in memory, he significantly reduced the performance impact of correcting data dependencies.

Many processors's instruction sets, such Intel's SSE3 or IBM's Altivec, provide *Shuffle* instruction that allows computing data dependent shuffles of short vectors without resorting to looping. Given two vectors A and B of size l, $C = Shuffle(A, B)$ of length l

is defined as $C[i] = A[B[i]]$. If $B[i] \geq l$, $C[i]$ is either undefined or some predefined value. Necessarily, l is relatively small as the vectors must be loaded into a register. In the following description $l = 16$, consistent with Intel's implementation of the instruction.

Rognes [58] introduces SWIPE – a vectorization of Smith-Waterman algorithm for aligning sets of protein sequences to the reference. Instead of computing a single Smith-Waterman matrix at a time, SWIPE computes l matrices simultaneously, where l is vector size of the processor. For each alignment the computation mimics a scalar Smith-Waterman algorithm: for k'th matrix, $k \in [1 \dots l]$, the algorithm uses only elements at position k of the vectors, thus avoiding data dependencies between different vector elements. For each position i on the reference S and position j on the query sequences r_k, $k \in [1 \dots l]$, the algorithm requires computing of *substitution score vector*: a vector Q of length l, s.t. $Q[k]$ is the substitution score between $r_k[j]$ and $S[i]$. The substitution vector is computed using *Shuffle* instructions available for Intel's SSE3: given two vectors A and B *Shuffle* allows computing a vector $C = Shuffle(A, B)$ of length l defined as $C[i] = A[B[i]]$. SWIPE computes all alignments with the same reference sequence and can not used to compute *multiple pairwise* alignments of protein sequences. The single reference limitation is essential to make possible efficient computation of substitution score vectors.

1.3.7 Mapping Read Pairs to a Reference

During two-pass DNA sequencing, such as by a SMS Helicos [21] sequencer, two reads are produced from every fragment of DNA. Both of the reads may have sequencing errors, the most common of which are skipped letters. These errors are nearly ten-fold more common than mis-calls or insertions.

Given unlimited computational resources, the best algorithm for mapping two reads sampled from the same location to the reference genome is full three-way alignment. This algorithm would require running time proportional to the product of the sequence lengths. Because the reads are typically short (∼30bp), the overall running time to map a single read pair to a genome of length n may be practical ($30 * 30 * n = 900n$), however the naive algorithm will not scale to aligning millions of reads that an SMS platform produces every day.

An alternative approach, we call it $PROFILE$, suggested in [21], is to align the two reads to each other, thus forming a profile, which could then be aligned to the reference genome. This approach, akin to the standard "progressive" alignment approaches used, e.g., in CLUSTALW [13], has the advantage of significantly decreased runtime, as the running time becomes proportional to the product of the lengths of the genome and the longer read (∼30 times the length of the genome), however, because the profile only incorporates a single optimal alignment, it loses important information about possible co-optimal and suboptimal alignments between the sequences. While the total number of optimal alignments between two sequences could be exponential in their lengths, Naor and Brutlag [51] and Hein [32] have suggested that all of these alignments can be represented compactly using a directed acyclic graph (DAG). Furthermore, Schwikowski and Vingron [60] have shown how to generalize the standard sequence alignment paradigms to alignment of DAGs, which they call weighted sequence graphs. Their original algorithm

Figure 1.1: Isolating a CA-repeat. Solid rectangle represent isolated fragment. Here $d = |CA| = 2$, the number of repetitions is $n = 3$ and the total length of the flanks for repetitions of sequence CA is $j = 3$. The total length of the fragment is $l = nd + j = 9$.

was stated in the context of the tree alignment problem, but is easily generalizable to any progressive alignment scoring scheme. Here we reintroduce the idea of representing alignments as graphs, and extend it for the SMS mapping problem.

1.4 Approximation Algorithm for Coloring of Dotted Interval Graphs

1.4.1 Biological Motivation

Microsatellite Genotyping

Genotyping is the process of determining genetic variation of an individual: given a DNA sample and a set of possible polymorphism (genetic variants) the goal is to find the specific variant (allele) the sample contains.

Microsatellites, also known as Short Tandem Repeats, are repetition of short DNA sequences, typically 2-6 bases long [65]. The number of repetition of various microsatellites is an important type of genetic polymorphism and used as a marker in clinical settings for diagnostics, for effective treatment selection as well as to evaluate for predisposition to a disease. They are also widely used in forensics, in kinship studies and for parentage assignment.

Microsatellite Assays

While it is possible to use sequencing methods for microsatellite assays, full sequencing of an individual genome is still relatively expensive. Hence, specialized assays are used for studying microsatellite polymorphism which are briefly described below. For further discussion of microsatelite genotyping we refer the reader to Zane et al.[74].

Typically for a site containing a sequence s of length d repeated n times (i.e. s^n) the first step of microsatellite assay involves isolating the repeating sequence s^n together with some flanks of total length j, see Fig 1.1. The flanks depend on the sequence s but not on the number of its repetitions. In the next step the fragment's length l is determined and the number of repetitions is reconstructed as $n = \frac{l-j}{d}$.

Microsatellite Assays – Multiplexing

To reduce costs of the assay several microsatellite sites can be studied in the same experiment. Two sites of repeats of microsatellite sequences can be studied simultaneously if

the length of corresponding isolated fragments are different. To minimize costs, given a set microsatellite repeat sites, we are interested to split the set to the minimum number of subsets such that for each subset all lengths of the isolated fragments corresponding to the sites are different. Next we show that this problem is related to the problem of coloring of a special class of interval graphs.

1.4.2 Coloring of Dotted Interval Graphs

Graph Coloring

The coloring of an undirected graph is an assignment of colors to graph's vertices such that no edge has both endpoints colored in the same color. The problem of graph coloring in minimum possible number of colors is NP-complete, moreover, it was proved by Lund and Yannakakis [44] that unless $P{=}NP$ there is no approximation algorithm for the graph coloring with performance guarantee N^{ϵ}, where N is the number of vertices of the graph and $0 < \epsilon < 1$.

Interval Graph

An interval graph is the intersection graph of a set intervals on the real line. Formally, given a set $I = \{[a_i, b_i]\}$ of closed intervals, a corresponding *interval graph* $G = (V, E)$ has a vertex set $V = I$ and an edge set

$$E = \{(I_1, I_2) | (I_1, I_2 \in I) \wedge (I_1 \cap I_2 \neq \emptyset)\}$$

There exists a polynomial time algorithm by Booth et al. [7] for recognition of interval graphs. Interval graphs were studied extensively, an introduction to the subject can be found in Golumbic [28].

Multiple-Interval Graph

Multiple-interval graph is a natural generalization of interval graph where each vertex may have more then one interval associated with it. Butman et al. [10] consider Minimum Vertex Cover, Minimum Dominating Set and Maximum Clique problems on multiple-interval graphs. Let t be the number of intervals associated with each vertex. Butman et al. [10] gives an approximation algorithm with ratio $2 - 1/t$ for Minimum Vertex Cover, t^2 for Minimum Dominating Set and $(t^2 - t + 1)/2$ for Maximum Clique.

Dotted Interval Graphs

A *Dotted Interval Graph* is a special case of multiple-interval graph introduced by Aumann et al. [3]. Its vertices are arithmetic progressions $I(j, d, k) = \{j, j + d, \ldots, j + dk\}$ where j, d and k are positive integers. Two vertices are connected by an undirected edge if the corresponding progressions intersect. For a dotted interval graph $G = (V, E)$ we say $G \in DIG_D$ if for any arithmetic progression $I(j, d, k) \in V$ it holds that $d \leq D$. It was proved by Aumann et al. [3] that the problem of optimal coloring of DIG_D graphs is NP-complete for $D = 2$.

Aumann et al. [3] shows an algorithm with approximation ratio $7/8D + \Theta(1)$ for Minimum Coloring Problem of DIG_D. Jiang [37] improves the approximation ratio to $(\frac{59}{72}D + O(\log D)) + \frac{D}{12\chi}$, where χ is the size of the optimal coloring. Yanovsky [70] subsequently improves the ratio to $\frac{2D+4}{3}$, see Chapter 5 for details.

Apart from Minimum Graph Coloring, Hermelin et al. [33] shows that Maximum Independent Set, Minimum Vertex Cover and Minimum Dominating Set can be solved in polynomial time for DIG_D, when D is fixed, solving an open problem posed in Jiang [37].

Relationship To Microsatellite Genotyping

Given a set of microsatellite sites let us define a dotted interval graph $G = (V, E)$ as follows. For a site containing repetitions of sequence s with corresponding flanks of total length j, we create a vertex $I(j, |s|, D)$, where D is the maximum number of repeats of any site of the set. Consider a coloring of graph G. By definition of a legal graph coloring for any two sites assigned the same color, they cannot have identical fragment length, independently of the number of repeats, from 1 to D, of corresponding sites. Hence a set of sites corresponding to the vertices colored with the same color can be processed in one experiment.

1.5 Our Contribution

1.5.1 Sequencing Dataset Compression

In Chapter 2 we address the problem of efficient storage of sequencing datasets. We design an algorithm, ReCoil [71], suitable for compression of very large datasets of short reads. While bandwidth of modern hard disk arrays can approach that of internal memory, their access times are several orders of magnitude slower. Hence, if the algorithm is designed in a way to minimize the number of random accesses, its performance can become competitive to the RAM-based algorithms.

ReCoil uses the natural idea that if two strings s_1 and s_2 are sufficiently similar, then it is more space efficient to store one of the strings and the differences between it and the second string, than to store both strings explicitly. Similarly to Coil [67], ReCoil makes use of a weighted *similarity graph* with the vertices corresponding to the sequences of the dataset. The set of edges joins pairs of similar sequences with edge weight reflecting the edit distance between the endpoints. The main contribution of ReCoil is overcoming the necessity to split large datasets into smaller chunks in order to fit in RAM.

We compare ReCoil to other compression programs and show that it achieves significantly higher compression and competitive runtimes. *Related to published work*: [71].

1.5.2 Alignment of Multipass Reads

Single Molecule Sequencing technologies such as Heliscope simplify preparation of DNA for sequencing, while sampling millions of reads in a day. Simultaneously, the technology suffers from a significantly higher error rate, compared to PCR-based methods, ameliorated by the ability to sample the same read more than once.

In Chapter 3 we introduce a novel rapid alignment algorithm for multi-pass single molecule sequencing methods. While we use the most common case of two-pass sequencing to describe our algorithms, they can be easily extended to a larger number of passes.

Our algorithm is based on the intuition that in a high-scoring three-way alignment of a pair of reads, S_1 and S_2, to a reference sequence, the alignment of sequences S_1 and S_2 to each other should also be high-scoring.

We combine the Weighted Sequence Graph (WSG) [60] representation of all optimal and near optimal alignments between the two reads sampled from a piece of DNA with k-mer filtering methods [56] and spaced seeds [11] to quickly generate candidate locations for the reads on the reference genome. We also propose a fast implementation of the Smith-Waterman algorithm using vectorized instructions that significantly speeds up the matching process. Our method combines these approaches in order to build an algorithm that is both fast and accurate, since it is able to take complete advantage of both of the reads sampled during two pass sequencing.

Our algorithms are implemented as part of the SHort Read Mapping Package (SHRiMP). SHRiMP is a set of tools for mapping reads to a genome, and includes specialized algorithms for mapping reads from popular short read sequencing platforms, such as Illumina/Solexa and AB SOLiD.

Related to published work: [72].

1.5.3 Parallel High Volume Mapping of Long High Error Rate Reads

In Chapter 4 we develop a fast seed-based mapping algorithm suitable for mapping of reads with high number of errors. The underlying data structure behind our algorithm is a fast and memory efficient index of the reference. Most existing DNA mapping tools construct a memory efficient index of a long DNA sequence based on one of the space efficient representations of the suffix tree. This construction is computationally intensive and time consuming, as result the reference is indexed in advance.

We propose a fast and memory efficient algorithm that, given a read r, allows us to quickly filter out the regions of S that are unlikely to contain a match of r. The idea is to use a hash-based postings lists intersection for the inverted list storing all k-mers in the reference. We use the fact that our problem is *offline* to reorder the tasks to improve performance, while keeping the index memory efficient – the actual inverted lists are never stored in memory.

In order to make index construction efficient, we develop an efficient algorithm to transpose a compact representation of a matrix of bits. This transposition algorithm is used in order to avoid cache inefficiencies due to long memory jumps between accesses. Fast index construction makes it possible to index the reference at the time of mapping, instead of building the index ahead of time. One of the advantages of not requiring constructing an index in advance is more efficient mapping of a dataset of reads against a subrange of the reference as only the parts of the reference that are of interest appear in the index, avoiding the step of filtering out the matches outside of the range of interest.

In addition, we develop a novel fully vectorized implementation of Smith-Waterman

algorithm to compute a set of pairwise alignments between two sets of sequences. This algorithm is used to chain exact matches found by the index using vector processing while avoiding data dependencies described in Chapter 4.

1.5.4 Approximation Algorithm for Coloring of Dotted Interval Graphs

In Chapter 5 we consider a problem of coloring of DIGs and improve the approximation ratio of Jiang [37] and Aumann et al. [3].

For $G \in DIG_D$ we provide a simple greedy algorithm for coloring of G with approximation ratio of $\frac{2D+4}{3}$. In order to prove the complexity bound we simplify and generalize the upper bound on the number of non-attacking queens on a triangular board considered by Nivasch and Lev [54]. *Related to published work*: [70].

Chapter 2

External Memory Compression of Sequencing Data

2.1 Introduction

2.1.1 The Problem

In this chapter we address the problem of compression of datasets of High Throughput Sequencing (HTS) reads without relying on availability of a reference sequence.

Datasets produced by modern HTS methods are usually of high *coverage*, i.e. they can have many different reads overlapping at each position of the genome, making them highly compressible. Previous research [12] show that for the sequence of a human genome it is hard to achieve compression rate significantly better than a trivial two bits per nucleotide. Hence the algorithms for compression of HTS datasets must take advantage of the self-similarity due to read overlaps. One difficulty that must be overcome is that for huge HTS datasets similar or overlapping reads can be at great distance from each other in the input and splitting it into smaller chunks will miss these similarities. Hence our goal in this chapter is a compression algorithm that works on the whole dataset at once, using external memory without a significant hit in performance.

2.1.2 Our Contribution

In our work we design a compression algorithm suitable for very large datasets. As for this task the internal memory is the main bottleneck, *ReCoil* does not assume that the input or the data structures created by the algorithms fit in RAM. Instead, *ReCoil* uses hard disks as its working memory. While bandwidth of modern disks is comparable to that of internal memory, their access times are many orders of magnitude slower. Hence, if the algorithm is designed in a way to minimize the number of random accesses, its performance can become competitive to the RAM-based algorithms. All steps of *ReCoil* were designed as reductions to either scanning or sorting of the input – two tasks that can be done I/O-efficiently. Decompression can be reduced to input scanning, hence it is very fast.

The main contribution of the ReCoil algorithm is overcoming the necessity to split large datasets into smaller chunks in order to fit in Random Access Memory. In addition, while sharing the idea of spanning tree based compression with *Coil*, our algorithm improves over *Coil* in various ways:

- ReCoil makes use of highly repetitive k-mers. Coil does not compress high count repeats, storing them explicitly.
- ReCoil's encoding scheme allows for encoding of matches of arbitrary length, while the only edit operations allowed by Coil are insertions, deletion and substitution of a single nucleotide. While SNPs are the most frequent mutation, Coil's encoding scheme is not efficient for encoding of the similarity due to overlaps between reads, which are the main source of compressibility of HTS datasets.
- ReCoil uses effectively the complementarity of the DNA in the dataset.

2.2 Methods

ReCoil uses the natural idea that if two strings s_1 and s_2 are sufficiently similar, then it is more space efficient to store one of the strings and the differences between it and the second string, then to store both strings explicitly.

2.2.1 Memory Model and Basic Definitions

Memory Model

We use the standard model of Aggarwal and Vitter [2] for analysis of external memory algorithms. In this model the machine has Random Access Memory (RAM) of size M and a disk of unlimited size. The CPU can operate only on data that is currently in RAM. Data is transferred between RAM and the disk in blocks of B consecutive elements, where $B < M$. In this model performance of algorithms is measured in terms of number of accesses to the disk, reflecting an observation that runtimes of most disk-bound algorithms are dominated by disk accesses. Aggarwal and Vitter [2] prove that in this model scanning an array of size n has complexity $\Theta(Scan\ n) = \Theta(n/B)$ and sorting requires $\Theta(Sort\ n) = \Theta(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ accesses to the disk. It is common to express I/O performance of an algorithm on data of size n in terms of the *Sort* and the *Scan* complexity.

Basic Definitions

Before proceeding to the description of the algorithm let us introduce some basic definitions:

- For an integer k and a string s we call the set of all substrings of s of length k the *seeds* or k-mers contained in s.

- Maximal exact matches (MEMs) are exact matches between two strings that cannot be extended in either direction towards the beginning or end of two strings without allowing for a mismatch.

Next we will give a brief overview of the compression algorithm and explain it in more detail later.

2.2.2 Overview

To encode the similarities between two reads ReCoil uses the set of MEMs between them: in order to encode s_2 given s_1 we store the locations of MEMs shared by s_1 and s_2, and the letters of s_2 not in these MEMs.

The similarity graph for the dataset is defined as a weighted undirected graph with vertices corresponding to the reads of the dataset. For any two reads s_1 and s_2 the weight of the edge connecting them should reflect the profitability of storing s_1 and the differences between it and s_2 versus storing both reads explicitly.

The encoding algorithm has four major steps, all of which have at most $O(Sort\ n)$ I/O complexity:

- Compute the similarity graph.
- Find the set of encoding edges – the edges of a maximum spanning tree (MST) in the similarity graph (there may in general be multiple MSTs).
- Pick the root of the MST arbitrarily and store the sequence in the root explicitly.
- Traverse the MST, encoding each node using the set of the Maximum Exact Matches (MEMs) between the node's read and the read of its parent in the MST.

Note that while the similarity graph is defined here as a clique, unprofitable edges of high weight are not used in the encoding and may be excluded from the graph. In addition, for simplicity of the MST construction we make the graph connected by adding for each i an edge of high weight between reads r_i and r_{i+1}.

2.2.3 Construction of the Similarity Graph

We define the weight of an edge of the *similarity graph* to be equal to the number of k-mers shared by the reads corresponding to the edge's endpoints. As for large datasets this graph cannot be held in memory, its construction was reduced to an external memory *merge sort* algorithm. In order to do this ReCoil creates an array containing the seeds for all reads, sorts this array by numeric representation of the seeds and uses the sorted array to create an anchor for each pair of reads sharing a seed:

This algorithm is illustrated in Fig. 2.1.

1. For each string S – a read or its reverse complement – generate all seeds contained in S.

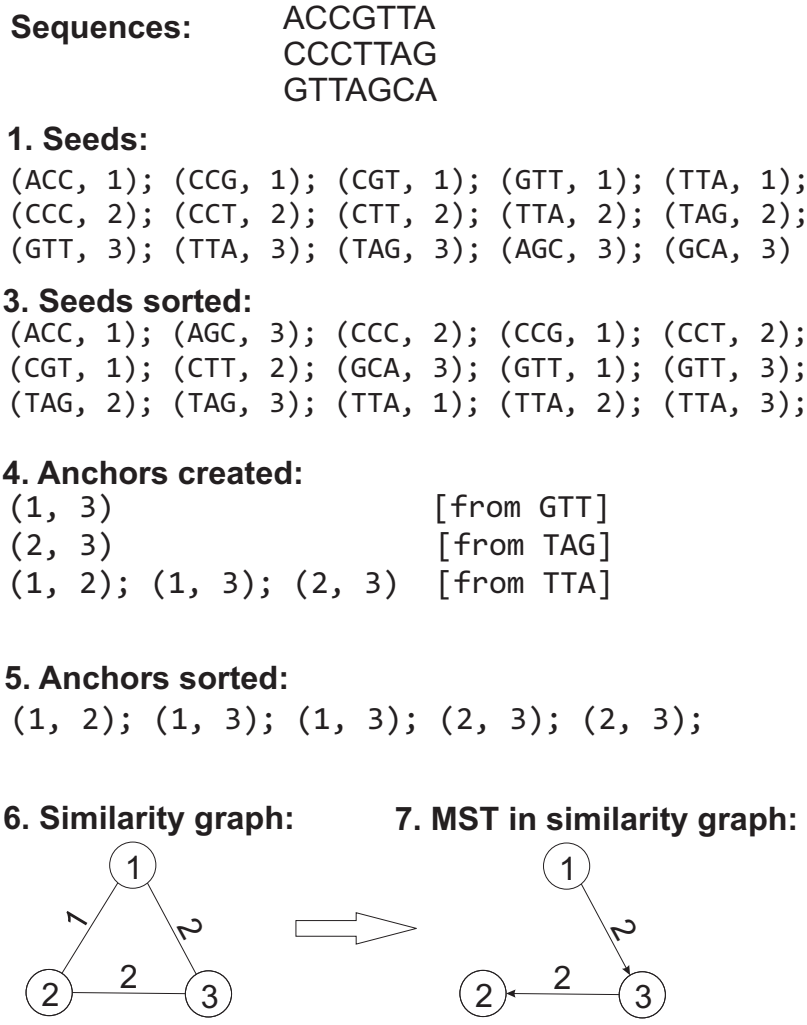

Figure 2.1: Top level illustration of the algorithm of Section 2.2.3 for construction of the *similarity graph* for three sequences. To simplify the illustration we do not consider here the reverse complement reads and the filtering of Step 2. The encoding that corresponds to the tree can be found in Section 2.2.4.

2. For some parameter t for each read select only the seeds with the t topmost numeric representations among all the seeds generated for that read. Output all selected seeds to a file.

3. Sort the file created in step 2 by numeric representation of the seeds.

4. Create an *anchors* file containing pairs of reads sharing a seed. (Below we describe a heuristic to deal with highly repetitive seeds.)

5. Sort the anchors file lexicographically using external memory sort.

6. Define the *similarity graph* as an undirected weighted graph with vertices corresponding to the reads and weight of an edge connecting two reads defined as the number of anchors corresponding to the pair of reads. (Note that after the sorting in step 5 the range of anchors corresponding to each pair of reads is contiguous.)

7. Use an external memory MST construction algorithm [18] in order to find the Maximum Spanning Tree in the similarity graph built in the previous step.

In step 2 we use the heuristic introduced in the Minimus Assember [63] in order to limit the number of seeds created by the algorithm.

Note that step 4 of the algorithm is quadratic in the number of reads sharing a seed. Fortunately, if the seeds are long, most of them have relatively low counts. Yet, to restrict the time ReCoil spends on each seed, for the range of reads containing the seed the algorithm first adds the anchors corresponding to the reads that are adjacent in the range, then those with distance of two between them, etc. until some predefined number of anchors per seed were created. In our tests the cut off was set to 80, i.e. we created at most 80 anchors for each seed.

The external memory MST stage has $O(Sort\ n)$ I/O complexity [18], where n stands for the number of edges in the similarity graph. It is easy to see that all other stages of the algorithm are reductions to either scanning or sorting of their inputs, thus have either $O(Scan\ n)$ or $O(Sort\ n)$ I/O complexity, where n is the size of the input to the stage.

In the following section we explain how we use the similarities between the reads for compression.

2.2.4 Encoding the Dataset

The external memory Kruskal algorithm of step 7 builds an unrooted tree where neither head-to-tail direction on the edges nor any traversal of the spanning tree are computed by the algorithm. Prim's algorithm results in a rooted tree but cannot be implemented efficiently in external memory. One solution to find the directions of edges (*rooting*) of the tree would be to use the external memory Breadth First Search (BFS) algorithm described in [8]. This algorithm computes the BFS order on a tree of N vertices in $O(Sort\ n)$ I/O operations using a reduction to an external memory algorithm for computing an Eulerian Tour in a graph. Yet while the similarity graph can be very large, its spanning tree can be stored in the RAM of a modern computer for any practical number of reads (vertices).

Hence we can use a simple in-memory BFS algorithm for rooting of the spanning tree as it is faster than the external memory BFS for graphs that can fit in RAM.

After the BFS, the algorithm uses an external memory sort to reorder the reads in the order of the BFS traversal of the MST. Let us denote with S the BFS-reordered read array and with $p(i)$ denote the position of a parent of read $S[i]$ in S. By construction, $i < j \Rightarrow p(i) \leq p(j)$.

Algorithm 1 scans through the array of BFS edges $(p(i), i)$ and S in parallel in order to encode read $S[i]$ based on read $S[p(i)]$. The algorithm runs several sequential scans simultaneously. In case of a single disk, k simultaneous scans can be implemented using a separate buffer of size B for each scan, where $B < M/k$. As a result, the I/O complexity of Algorithm 1 is the same as of the scan: while the scan can use any block size $B < M$, the only difference would be in the multiplicative constant. In the description below, $Load(destRAM, srcdisk, pos)$ buffers into a RAM array $destRAM$ at most B bytes, starting from position pos of a disk array $srcdisk$. The value returned by the function is the position of the last element read.

Algorithm 1 Encoding of a BFS-ordered sequence S of reads.

Init: Allocate arrays $Children$ and $Parents$ of B bytes.
Let $LastParent = LastChild = 0$, $i = 1$
Write down $S[0]$ – the sequence in the root – explicitly.
for all BFS edges $(p(i), i)$ **do**
  if $i > LastChild$ **then**
    $LastChild = Load(Children, S, i)$
  end if
  if $p(i) > LastParent$ **then**
    $LastParent = Load(Parents, S, p(i))$
  end if
  Use the MEM-based encoding of Section 2.2.5 to encode the sequence $S[i]$ given $S[p[i]]$ or the reverse complement of $S[p[i]]$, depending on what results in a shorter encoding.
end for
Finalize: Use a general purpose compression algorithm, such as *7-zip*, to further improve the compression.

To illustrate the encoding, for the sequences of Fig. 2.1 we have:

1. Reorder the sequences in the BFS order of the MST, rooted in sequence 1, i.e. to (1, 3, 2): *ACCGTTA, GTTAGCA, CCCTTAG*. Let's label the reordered sequences as 1', 2', 3'.

2. Write the root explicitly: *ACCGTTA*.

3. Encode 2' referring to 1': ReadLen = 7, *MEMs = (SourceID = 1', StartInSource = 3, StartInDest = 0, Length = 4)*; $PlainTextLetters = GTT$.

4. Encode 3' referring to 2': ReadLen = 7, *MEMs = (SourceID = 2', StartInSource = 2, StartInDest = 3, Length = 4)*; $PlainTextLetters = CCC$.

To further improve the compression rate, ReCoil uses difference coding to reduce the ranges of the numbers it stores. Like Coil, it uses separate files for storing properties such as read lengths as these values are typically identical and can be compressed well using the final general purpose compression step. The combined I/O complexity of this stage is the same as that of external memory sorting of the reads.

Finally, we show how ReCoil finds the MEM-based encoding of one read relative to another similar read.

2.2.5 Encoding Sequence Similarities

Given two strings s_1 and s_2 we use the following simple algorithm to find all MEMs shared by them:

1. Merge the seeds contained in s_1 and s_2 in one array and sort this array using the seeds' numeric representations as the keys.

2. Scan the sorted array and for each seed shared by both s_1 and s_2 create a tuple (r_1, r_2), where r_1 and r_2 are positions of the seed in s_1 and s_2 respectively.

3. Sort the tuples defined above lexicographically, where the keys are defined as $(r_1 - r_2, r_1)$, i.e. first by diagonal of the match corresponding to the anchor, then by the position of the match in the first read; as a result, the anchors corresponding to each MEM follow sequentially, making it easy to extract the MEMs. A similar sorting strategy was used by Ning et al. [53].

In the next step ReCoil uses the sparse dynamic programming algorithm of Eppstein et al. [19] in order to find the subset of MEMs that results in the optimal compression of s_1 relative to s_2. This algorithm finds the optimal alignment between two sequences under affine gap cost, subject to the restriction that each matching segment is at least k nucleotides long. The Gap Open Penalty is defined by the shortest MEM length that is profitable to use for the encoding. For larger values of k this algorithm is much faster than the Smith-Waterman [64] algorithm.

Denote by $Gain(s_1, s_2)$ the space saved if s_2 is encoded relative to s_1 using the encoding described above. Since for every MEM in the encoding, the space saved is the difference between the memory required to store the location of the MEM versus storing it explicitly, *Gain* function is symmetric and the same amount of space is saved, whether s_1 is encoded relative to s_2 or vice versa.

2.2.6 Decompression

The inputs to the decompressor are several streams produced by the encoder: the read lengths, the list of MEMs shared between the reads and their parents that were used for the encoding, as well as explicitly stored letters of the reads. These streams are read

Figure 2.2: Comparison of compression with ReCoil, Coil, 7-zip and bzip2 for various coverages. The simulated datasets were generated by making random samples of length 70 from Human Chromosome 14, adding single-nucleotide errors (insertions, deletions or substitutions) with probability 0.02 and reverse complementing each read with probability 0.5.

concurrently. Note that the data in the streams is ordered in the BFS order on the encoding spanning tree. As in the case of algorithm 1, the decoding uses several buffers in order to implement multiple sequential accesses to a single disk.

Since the reads are decoded in the BFS order on the encoding tree, in order to decode, we create a stream of the decoded reads maintaining the position of the parent of the currently decoded read. Since this position changes in a non-decreasing order, the decompression step has $O(Scan\ n)$ I/O complexity, i.e. $O(n/B)$ disk accesses, where B is the size of the block loaded into memory on each access.

2.3 Results and Discussion

ReCoil was implemented in C++ using the STXXL [17] library of external memory data structures. The k-mer size selected was 15 for the similarity graph construction and 10 for the encoding step. We tested ReCoil on both simulated and real datasets. The simulated datasets were generated by making random samples of given length from Human Chromosome 14, adding single-nucleotide errors (insertions, deletions or substitutions) with probability 0.02 and reverse complementing each read with probability 0.5. All our generated datasets were of the same size of 1.8 billion nucleotides.

The results are summarized in Table 2.1 and Figure 2.2. From the results on the synthetic datasets we can see that as the read length increases, the compression rate of ReCoil improves. One explanation for the better compression is that longer reads are more likely to share more and longer MEMs. Also in our tests the run times were decreasing when the length of the synthetic reads was increasing: if the coverage is kept constant, longer reads result in smaller similarity graphs. On the other hand the general purpose algorithms have runtimes and compression rate independent of read length.

		ReCoil			Coil			7-zip			bzip2		
Readlen	Megabases	Size	Time	%	Size	Time	%	Size	Time	%	Size	Time	%
36	6912	1180	840	0.17	NA	NA	NA	1900	300	0.27	2250	45	0.36
70	1800	326	290	0.18	450	650	0.25	412	78	0.23	483	11	0.27
100	1800	278	246	0.15	415	625	0.23	405	76	0.22	481	11	0.27
120	1800	241	198	0.13	387	590	0.21	403	77	0.22	480	11	0.27

Table 2.1: Comparison of compressed file size, runtime and compression ratios of ReCoil to Coil, 7-zip and bzip2. File sizes are shown in megabytes and run times in minutes. ReCoil, Coil and 7-zip were given 2 GB of RAM, while bzip2 was run with the compression-level=9. Both ReCoil and Coil used 7-zip as a post-processing compression step, this step was included in the timings. Decoding time for ReCoil was less than 5 minutes for all datasets, including the 7-zip uncompress step.

There are several reasons why we were able to achieve better compression rate than Coil, 7-zip and bzip2. First, the algorithms we implemented in ReCoil allow us to make

use of similarities between the reads located far from each other in the input, as ReCoil does not require splitting of the input into smaller parts. Another reason for better compression of ReCoil than Coil, is the fact that the edges between the reads in our similarity graph reflect better the gain obtained by encoding one read relative to another.

To test ReCoil on a real short read data, we compressed a dataset of 192 million Illumina reads of length 36 downloaded from `http://www.ncbi.nlm.nih.gov/sra/SRX001540`, which is a part of "Human male HapMap individual NA18507" project (`http://www.ncbi.nlm.nih.gov/sra/SRP000239`). This resulted in a file of size 1.18 GB. *7-zip* compressed the same sequences to size 1.9 GB. *7-zip* was our general purpose compressor of choice in the comparisons since in our experience it resulted in the best compression. Out of 192 million sequences in the dataset, ReCoil stored 4 million sequences explicitly, the rest were stored compressed. We were not able to run Coil on this dataset, while it took about 14 hours for ReCoil to compress this dataset using a 1.6GHz Celeron with four hard disks and 4 GB of RAM. ReCoil can make use of several disks installed on the machine not only to scale up to large datasets but also to speed up the computations due to higher disk bandwidth. Nevertheless all the algorithms remain efficient if only a single hard disk is present.

We also attempted to compare our algorithm to a publically available reference-based compression algorithm MZip [35]. Unfortunately it could not scale to the size of our datasets as just its preprocessing step, converting the results of a BWA mapping program to its internal format took more than 90 minutes on a set of 3.5 million reads of length 36, not counting the BWA alignment step itself, and we were unable to run the pipeline to completion.

2.4 Limitations and Future Work

Our goal in this work was to design an algorithm that maximizes compression ratio. Yet the ability to retrieve a sequence without full decompression will improve applicability of the algorithm. One simple way to accomplish this would be to store the sequences at some levels of the spanning tree explicitly. Then in order to decode a sequence of a node one must go up in the tree until reaching a level of not encoded nodes.

One limitation of ReCoil in practice is that it only deals with the sequencing data, while compressing the metadata such as quality values is as important. Another limitation of ReCoil's approach is that in order to limit the number of anchors corresponding to repeats and to reduce the number of seeds created we had to use various heuristics that compromise how well the weights of the edges of the similarity graph reflect the savings from encoding one read relative to another. Ferragina et al. [24] describe an algorithm for computing the BWT transform in external memory. While we saw in our testing that BWT-based *bzip2* provided significantly worse compression than ReCoil, one direction for future work would be to use a BWT-based approach in order to find the MEMs and build the similarity graph more efficiently.

Chapter 3

Alignment of Multipass Reads

3.1 The Problem

In this chapter we develop novel rapid alignment algorithms for two-pass Single Molecule Sequencing methods. We combine the Weighted Sequence Graph (WSG) representation of all optimal and near optimal alignments between the two reads sampled from a piece of DNA with k-mer filtering methods and spaced seeds to quickly generate candidate locations for the reads on the reference genome. We also propose a fast implementation of the Smith-Waterman algorithm using vectorized instructions that significantly speeds up the matching process. Our method combines these approaches in order to build an algorithm that is both fast and accurate, since it is able to take complete advantage of both of the reads sampled during two pass sequencing.

3.2 Algorithms

In this section we describe our algorithm for mapping two reads generated from a single DNA sequence with SMS to a reference genome. We start (Sections 3.2.1 and 3.2.2) by reviewing how the Weighted Sequence Graph data structure [60] can be used to speed up the inherently cubic naive alignment algorithm to near-quadratic running time. We will then demonstrate how to combine the WSG alignment approach with standard heuristics for rapid alignment, such as seeded alignment and k-mer filters in Section 3.2.3. Finally, we present our implementation of the Smith-Waterman alignment algorithm with SSE vector instructions to further speed up our algorithm, making it practical for aligning millions of reads against a long reference genome.

3.2.1 Alignment with Weighted Sequence Graphs

Given unlimited computational resources, the best algorithm for mapping two reads sampled from the same location to the reference genome is full three-way alignment. This algorithm would require running time proportional to the product of the sequence lengths. Because the reads are typically short ($\sim$30bp), the overall running time to map a single read pair to a genome of length n may be practical ($30 * 30 * n = 900n$), however

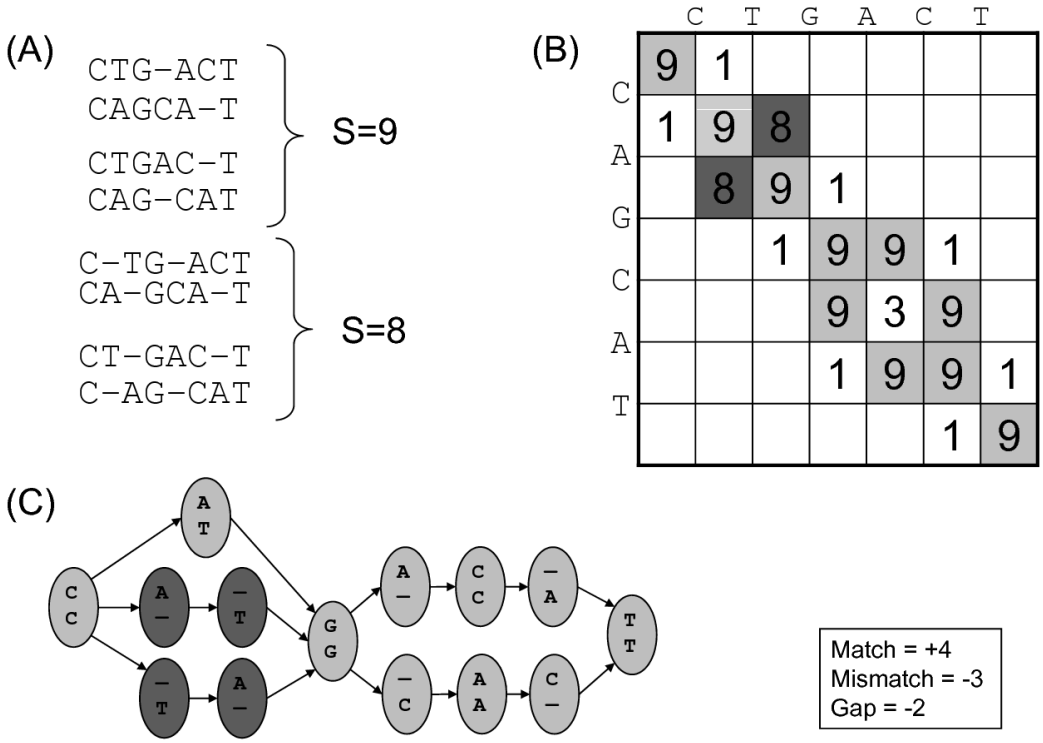

Figure 3.1: An example of a WSG graph representing 1-suboptimal alignments of CTGACT with CAGCAT. **(A)**. Optimal alignments of score 9 and two 1-suboptimal of score 8; two more 1-suboptimal alignments are possible. **(B).** Cells of the dynamic programming matrix such that the best path through them is 9 are shaded in light grey, if it is 8 – in dark grey. **(C).** WSG corresponding to at worst 1-suboptimal alignments.

the naive algorithm will not scale to aligning millions of reads that an SMS platform produces every day.

An alternative approach, suggested in [21], is to align the two reads to each other, thus forming a profile, which could then be aligned to the reference genome. This approach, akin to the standard "progressive" alignment approaches used, e.g., in CLUSTALW [13], has the advantage of significantly decreased runtime, as the running time becomes proportional to the product of the lengths of the genome and the longer read (∼30 times the length of the genome), however, because the profile only incorporates a single optimal alignment, it loses important information about possible co-optimal and suboptimal alignments between the sequences. For example, Figure 3.1 demonstrates two short sequences with two optimal alignments, as well as 4 more near-optimal ones. While the total number of optimal alignments between two sequences could be exponential in their lengths, Naor and Brutlag [51] and Hein [32] have suggested that all of these alignments can be represented compactly using a directed acyclic graph (DAG). Furthermore, Schwikowski and Vingron [60] have shown how to generalize the standard sequence alignment paradigms to alignment of DAGs, which they call weighted sequence graphs. Their original algorithm was stated in the context of the tree alignment problem, but is easily generalizable to any progressive alignment scoring scheme. Here we reintroduce the idea of representing alignments as graphs, and extend it for the SMS mapping problem.

Definitions

The following definitions will prove useful:

- Given the sequence S of length $|S|$ over an alphabet $\Sigma = \{A, C, T, G\}$, we refer to its ith symbol as $S[i]$

- We say that a sequence S' over alphabet $\overline{\Sigma} = \Sigma \bigcup \text{`} - \text{`}$, where character ` $-$ ` is called a gap, *spells* S if removing the gaps from S' results in sequence S. When this does not cause ambiguity, we will write S instead of S'.

- A k-mer is any sequence of length k.

- When addressing matrix indices, we use a notations "$i_1 : i_2$", "$: i_2$", "$i_1 :$" and "$:$" to represent the sets of indices j such that $i_1 \leq j \leq i_2$, $j \leq i_2$, $i_1 \leq j$ and all possible indices respectively. For example, given matrix M of size $K \times N$ we shall denote the i'th row of the matrix by $M[i; :]$ and its j'th column by $M[:; j]$.

- A *global alignment* of 2 sequences S_1 and S_2 over alphabet Σ is defined as a matrix $M = [2 \times N]$ such that $M[i; :]$ spells S_i for $i = 1, 2$.

- Given a real-valued score function $sc : \overline{\Sigma} \times \overline{\Sigma} \mapsto \Re$ we define the score SC of the alignment $M = [2 \times N]$ as

$$SC(M) = \Sigma_{j=1}^{N} sc(M[:; j])$$

- The *global alignment problem* is to find an alignment M such that $SC(M)$ is maximum. It is known [52] that for any cost function sc satisfying condition $sc(-,-) < 0$ the problem is well-defined though its solution is not necessarily unique. We denote the maximum score with Opt.

- We call an alignment M ε-suboptimal if $SC(M) \geq Opt - \varepsilon$.

While we defined the alignment for two sequences, the problem for more than two sequences is defined similarly. In particular, the problem we consider in this work – an alignment of a pair of sequences S_1 and S_2 to a reference sequence R – is to find a high scoring alignment matrix $M = [3 \times N]$ such that $M[i;:] = S_i$ for i=1 and 2 respectively and $M[3;:]$ is a substring of R.

Representing (Sub)-optimal Alignments in a Graph

A sequence S can be represented by a labeled directed graph $G'(S) = (V, E)$ with $|S|+1$ nodes $\{s = V[0] \cdots V[|S|] = t]\}$ and $|S|$ edges: for every $0 \leq i < |S|$ there is an edge $V[i] \to V[i+1]$ labeled with $S[i]$ – the i'th symbol of sequence S; vertices s and t will be called the source and the sink. We obtain the graph $G(S)$ from graph G' by adding a self loop edge $V[i] \to V[i]$ labeled with a gap symbol '–' for every i. There is a one-to-one correspondence between the sequences over alphabet $\overline{\Sigma}$ spelling sequence S and the sequences produced by reading edge labels along the paths in $G(S)$ from s to t.

Given two strings, S_1 and S_2 and two graphs $A = G(S_1)$ and $B = G(S_2)$, their cross product $A \times B$ is defined as a graph with the vertex set $V = \{(v_A, v_B)\}$, for any v_A vertex of A and v_B vertex of B, and edge set $E = \{v_1 \to v_2\}$, for any $v_1 = (v_A^1, v_B^1)$ and $v_2 = (v_A^2, v_B^2)$, such that there is an edge $v_A^1 \to v_A^2$ in A and an edge $v_B^1 \to v_B^2$ in B. The edges of all graphs considered in this work will be labeled with strings over alphabet $\overline{\Sigma}$; the edge $v^1 \to v^2$ of the cross product graph will receive a label which is the concatenation of the labels of an edge $v_A^1 \to v_A^2$ in A and an edge $v_B^1 \to v_B^2$ in B.

This cross product graph, sometimes referred to as an edit graph, corresponds to the Smith-Waterman dynamic programming matrix of the two strings. It is easy to prove that there is a one-to-one correspondence between the paths from *source* $s = s_1 \times s_2$ to *sink* $t = t_1 \times t_2$ and alignments: if the sequence of edges along the path is $e_0 \cdots e_{N-1}$, then the corresponding alignment M is defined as $M(:;j) = label(e_j)$ for all values of j – recall that $label(e)$ is a pair of symbols. Note that the only cycles in the cross-product graph we just defined are the self loops labeled with '– –' at every node of the graph.

Now a *WSG* is defined as a subgraph of the cross-product graph such that its vertex set V contains vertices s, t and for every vertex $v \in V$ it is on some path from s to t. Intuitively, one may think of the WSG as a succinct representation of a set of alignments between the two strings. We use WSGs in Section 3.2.2 for the purpose of representing all high-scoring alignments.

3.2.2 Alignment of Read Pairs with WSG

During two-pass DNA sequencing, two reads are produced from every fragment of DNA. Both of the reads may have sequencing errors, the most common of which are skipped

letters. These errors are nearly ten-fold more common than mis-calls or insertions [21]. Our algorithm for aligning two-pass SMS reads is based on the intuition that in a high-scoring three-way alignment of a pair of reads, S_1 and S_2, to a reference sequence, the alignment of sequences S_1 and S_2 to each other should also be high-scoring. The algorithm proceeds as follows:

- Build a cross-product graph $G' = (V, E)$ representing the set of all possible alignments of S_1 and S_2, s is the source of G', t is the sink.
- For every edge $e = u \to v$ in G' we compute the score of the highest scoring path from source s to sink t through the edge e; we denote this score as $w(e)$, the weight of the edge. To do this we use dynamic programming to compute the scores of the highest scoring paths from s to every vertex of G' and of the highest scoring paths, using the edges of G' reversed, from t to every vertex of G. Time complexity of this step is $O(E)$.
- For a given suboptimality parameter ε, we build WSG G_2 from G' by discarding all edges such that
$$w(e) < Opt - \varepsilon$$
where Opt denotes the score of the highest scoring path from s to t in G'. Observe that while all ε-suboptimal alignments of S_1 and S_2 correspond to paths from s to t, in G_2, the converse is not true: not all paths from s to t are ε-suboptimal.
- Align WSG G_2 to the reference genome: compute the cross product of G_2 and the linear chain that represents the reference sequence, obtaining a WSG spelling all possible three way alignments M, such that $M[1:2;:]$ is one of the alignments, represented by G_2. Finally, use dynamic programming (similar to Smith and Waterman [62]) to search this graph for the locally optimal path. The score of this path is the score of mapping the pair of reads at the given location.

While the use of the WSG leads to a significant speedup compared with the full 3-dimensional dynamic programming, this is still insufficient for the alignment of many reads to a long reference genome, as the resulting algorithm is at best quadratic. In the subsequent section we combine the WSG alignment approach with standard heuristics for rapid sequence alignment in order to further improve the running time.

3.2.3 Seeded Alignment Approach for SMS Reads

Almost all heuristic alignment algorithms start with seed generation – the location of short, exact or nearly exact matches between two sequences. Whenever one or more seeds are found, the similarity is typically verified using a slow, but more sensitive alignment algorithm. The seeds offer a tradeoff between running time and sensitivity: short seeds are more sensitive than long ones, but lead to higher running times. Conversely, longer seeds result in faster searches, but are less sensitive. Our approach for SMS seeded alignment differs in two ways from standard methods. First, we generate potential seeds not from the observed reads, but rather from the ϵ-suboptimal WSGs representing their

alignments. Second, we combine spaced seeds [11] with k-mer-filtering techniques [56] to further speed up our algorithm.

Seed Generation

Intuitively, given a read pair and an integer k, the seeds we want are k-long substrings (k-mers) likely to appear in the DNA segment represented by the pair and segment's matching location on the reference sequence. While k-mers present in each individual read may not match the genome directly due to the deletion errors introduced during the sequencing process, it is much less likely that both reads have an error at the same location. Consequently, we generate the potential seeds not from the the reads directly, but from the high-scoring alignments between them. Because every high-scoring alignment between the reads corresponds to a path through the WSG, we take all k-long subpaths in the WSG and output the sequence of edge labels along the path. Recall that each edge of the WSG is labeled by a pair of letters (or gaps), one from each read. If one of the reads is gapped at the position, we output the letter from the other read. If neither read is gapped we output a letter from an arbitrary read. While the number of k-mers we generate per pair of reads can be large if the two reads are very different, in practice the reads are similar and the WSG is small and "narrow". In our simulations (described in the Results section) we saw an average of 27 k-mers per read-pair for reads of approximately 30 nucleotides.

Genome Scan

Given the sets of k-mers generated from the WSGs of read-pairs, we build a lookup table, mapping from the k-mers to the reads. We used spaced seeds [11], where certain pre-determined positions are "squeezed" from the k-mers to increase sensitivity. We then scan the genome, searching for matches between k-mers present in the genome and the reads. If a particular read has as many or more than a specified number of k-mer matches within a given window of the genome, we execute a vectorized Smith-Waterman step, described in the next section.

Unlike some local alignment programs that build an index of the genome and then scan it with each query (read), we build an index of the reads and query this index with the genome. This approach has several advantages: first, it allows us to control memory usage, as our algorithm never needs memory proportional to the size of the genome, while the large set of short reads can be easily divided between many machines in a compute cluster. Secondly, our algorithm is able to rapidly isolate which reads have several k-mer matches within a small window by using a circular buffer to store all of the recent positions in the genome that matched the read. The genome scan algorithm is illustrated in Figure 3.2A.

3.2.4 Vectorized Smith-Waterman Implementation

While the genome scan described in the previous section significantly reduces the number of candidate regions, many false positives are encountered. To further reduce the num-

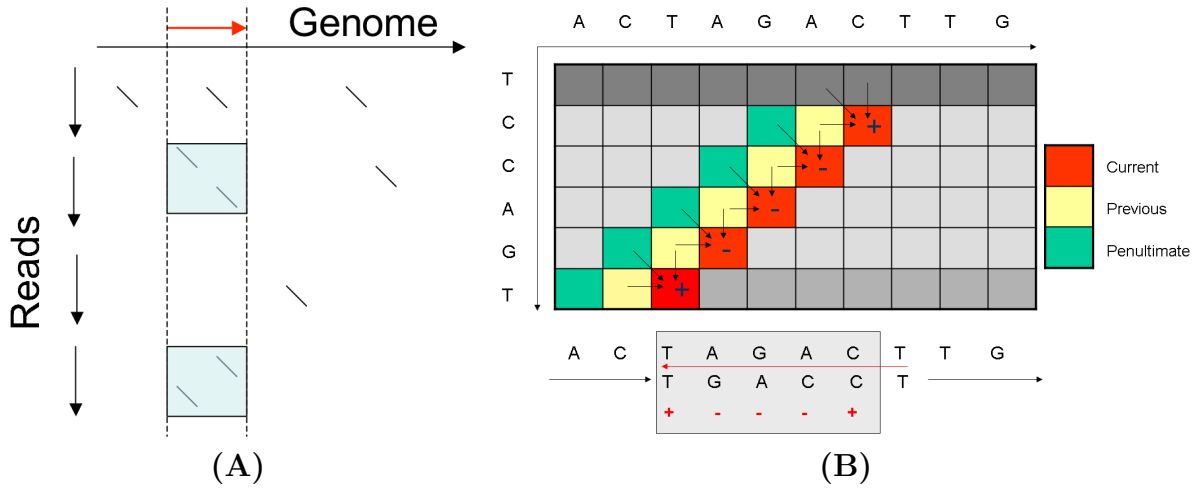

Figure 3.2: **A.** Overview of the k-mer filtering stage within SHRiMP: A window is moved along the genome. If a particular read has a preset number of k-mers within the window the vectorized Smith-Waterman stage is run to align the read to the genome. **B.** Schematic of the vectorized-implementation of the Smith-Waterman algorithm. The red cells are the vector being computed, on the basis of the vectors computed in the last step (yellow) and the next-to-last (green). The match/mismatch vector for the diagonal is determined by comparing one sequence with the other one reversed (indicated by the red arrow below). To get the set of match/mismatch positions for the next diagonal the lower sequence needs to be shifted to the right.

ber of potential mapping positions given to the WSG-based aligner, we first align both reads to the candidate region using a vectorized implementation of the Smith-Waterman algorithm.

We propose a method, where the running time of the fully vectorized algorithm is independent of the number of matches and mismatches in the matrix, though it only supports fixed match/mismatch scores (rather than full scoring matrices). Since SHRiMP only applies a vectorized Smith-Waterman scan to short regions of confirmed k-mer hits, alternative approaches that benefit by skipping computation in areas of dissimilarity are unable to take significant advantage. Figure 3.2B demonstrates the essence of our algorithm: by storing one of the sequences backwards, we can align them in such a way that a small number of logical instructions obtain the positions of matches and mismatches for a given diagonal. We then construct a vector of match and mismatch scores for every cell of the diagonal without having to use expensive and un-vectorizable load instructions or pre-compute a "query profile". In our tests (see Table 3.1), our approach surpasses the performance of Wozniak's original algorithm [69] and performs on par with Farrar's method [22]. The advantages of our method over Farrar's approach are simplicity, independence of the running time and the scores used for matches/mismatches/gaps, and linear scalability for larger vector sizes. The disadvantages of our method are that it cannot support full scoring matrices (i.e. it is restricted to match/mismatch scores) and is slower for queries against large reference sequences where significant areas of dissimilarity are expected. However, the former is less important for DNA alignment and the latter does not apply to SHRiMP.

The vectorized Smith-Waterman approach described above is used to rapidly determine if both of the reads have a strong match to a given region of the genome. The locations of the top n hits for each read on the genome are stored in a heap data structure, which is updated after every invocation of the vectorized Smith-Waterman algorithm if the heap is not full, or if the attained score is greater than or equal to the lowest scoring top hit. Once the whole genome is processed, the highest scoring n matches are re-aligned using the full WSG alignment algorithm described in Section 3.2.2.

Processor type	Unvectorized	Wozniak	Farrar	SHRiMP
P4 Xeon	97	261	335	338
Core 2	105	285	533	537

Table 3.1: Performance (in millions of cells per second) of the various Smith-Waterman implementations, including a regular implementation, Wozniak's diagonal implementation with memory lookups, Farrar's striped algorithm and our diagonal approach without score lookups (SHRiMP). We used each method within SHRiMP to align 50 thousand reads to a reference genome with default parameters. The improvement of the Core 2 architecture for vectorized instructions lead to a significant speedup for our and Farrar's approaches, while the Wozniak algorithm's slight improvement is due to the slow memory lookups.

3.3 Results

In order to test the efficacy of our read mapping algorithm we designed a simulated dataset with properties similar to those expected from the first generation SMS sequencing technologies, such as Helicos [21]. We sampled 10,000 pairs of reads from human chromosome one (the total length of the chromosome is approximately 1/10th of the human genome). The first read was sampled to have a mean length of 30 bases, with standard deviation of 8.9, the second one had a mean length of 26 bases with standard deviation 8.6. The lengths of both reads ranged from 12 to 58 bases. We introduced errors into each of the the reads, with 7% deletion errors and 0.5% insertion and substitution errors uniformly at random. We also introduced Single Nucleotide Polymorphisms (SNPs) at a rate of 1%.

We used three approaches to map the read pairs back to the reference. Our main approach was the WSG-based alignment algorithm described in Section 3.2.2. For comparison we also implemented two alternatives: mapping the reads individually, using the version of SHRiMP for Illumina/Solexa, but with the same parameters as were used for WSG-based mapping in order to properly capture the high expected deletion rates, and a profile-based approach, which follows the overall framework of the WSG approach, but only considers a single top alignment (one path in the WSG) for further mapping to the reference genome. The methods are labeled WSG, $SEPARATE$, and $PROFILE$, respectively, in Table 3.2.

We evaluated the performance of these algorithms based on three criteria: the fraction of the reads that were not mapped at all (lower is better), the fraction of the reads

that were not mapped uniquely (lower is better: while some reads have several equally good alignments due to repetitive DNA, others map with equal scores to non-repetitive DNA segments, and this should be minimized), and the percentage of the reads mapped uniquely and correctly (higher is better). When evaluating the two reads separately, we considered a hit unique if either of the two reads had a unique top hit, and we considered a unique top hit correct if either of the top hits was correct. We ran all algorithms with three seed weights: 8, 9, and 10, with each spaced seed having a single wildcard character (zero) in the middle of the seed (e.g. 1111101111 was the spaced seed of weight 9).

As one can see in Table 3.2, the approach of mapping the reads separately, while leading to very good sensitivity (almost all reads aligned), has poor specificity (only 70-78% of the reads have a unique top hit, with 61-64% both unique and correct). The WSG and $PROFILE$ approaches perform similarly, with the WSG approach slightly more sensitive and having a higher fraction of unique correct mappings for all seed sizes.

Type	SEPARATE			PROFILE			WSG		
seed weight	8	9	10	8	9	10	8	9	10
no hits %	**0.000**	**0.131**	**2.945**	1.741	4.905	10.52	1.609	4.314	10.15
multiple %	30.12	26.45	21.13	**10.20**	9.342	8.353	10.44	**9.127**	**8.258**
unique cor %	63.90	63.00	61.09	78.96	74.90	69.66	**79.17**	**75.84**	**70.85**
runtime	28m16	**8m48**	**4m22**	**27m17**	11m32	6m58	30m59	12m13	7m13

Table 3.2: Comparison of the WSG-based alignment with the two alternative approaches. The first two rows describe the percentage of reads not mapped anywhere on the genome or mapped in multiple places (with equal alignment scores). The third row is the percentage of the unique hits that are correct. The sub-columns under each method are the number of matching positions in the spaced seed (we allowed for a single non-matching character in the middle of the seed). The last row shows the running time in minutes and seconds on a 2.66 GHz Core2 processor. The best result in each category is in boldface.

3.4 Discussion

In this paper we propose a novel read alignment algorithm for next-generation Single Molecule Sequencing (SMS) platforms. Our algorithm takes advantage of spaced k-mer seeds and effective filtering techniques to identify potential areas of similarity, a novel, vectorized implementation of the Smith-Waterman dynamic programming algorithm to confirm the similarity and a weighted sequence graph-based three way final alignment algorithm. Our approach is implemented as part of SHRiMP, the SHort Read Mapping Package, and is freely available at **http://compbio.cs.toronto.edu/shrimp**.

While the overall emphasis of the current paper was read mapping, we believe that several of our methods may have broader applications. For example, our implementation of the Smith-Waterman algorithm can be used in other sequence alignment methods, and unlike previous vectorized implementations the running time of our algorithm is independent of the scoring parameters. The weighted sequence graph model could also

be useful for ab initio genome assembly of SMS data, where a variant of our method could be used for overlap graph construction.

Chapter 4

Parallel High Volume Mapping of Long High Error Rate Reads

4.1 The Problem

In this chapter we develop a fast seed-based mapping algorithm suitable for mapping of reads with high number of errors. The underlying data structure behind our algorithm is a fast and memory efficient index of the reference. Most existing DNA mapping tools construct a memory efficient index of a long DNA sequence based on one of the space efficient representations of the suffix tree or suffix array. This construction is computationally intensive, as result the reference is always indexed in advance.

We propose a fast and memory efficient algorithm that, given a read r, allows us to quickly filter out the regions of S that are unlikely to contain a match of r. The idea is to use a hash-based postings lists intersection for the inverted list storing all k-mers in the reference. We use the fact that our problem is *offline* to reorder the tasks to improve performance, while keeping the index memory efficient – the actual inverted lists are never stored in memory.

In order to make index construction efficient, we develop an efficient algorithm to transpose a compact representation of a matrix of bits. This transposition algorithm is used in order to avoid cache inefficiencies due to long memory jumps between accesses. Fast index construction makes it possible to index the reference at the time of mapping, instead of building the index ahead of time. One of the advantages of not requiring constructing an index in advance is more efficient mapping of a dataset of reads against a subrange of the reference as only the parts of the reference that are of interest appear in the index, avoiding the step of filtering out the matches outside of the range of interest.

In addition, we develop a novel fully vectorized implementation of Smith-Waterman algorithm to compute a set of pairwise alignments between two sets of sequences. This algorithm is used to chain exact matches found by the index using vector processing while avoiding vectorization-limiting data dependencies described in Chapter 4.

4.2 Related Data Structures

4.2.1 Inverted Lists

The index data structure we develop is similar in some ways to *inverted list*. A common query to a search engine involves finding documents containing a set of search terms. *Inverted list* is the dominant data structure leveraged by search engines to answer this query. *Inverted list*, also referred as *inverted file* or *inverted index* is an array with an entry corresponding to each possible term. An entry for a term is the list containing all documents where the term occurs. This list is called a *postings list* for the term. Given a set of terms, the search engine must intersect the postings lists associated with the terms of the set.

While inverted lists were researched extensively in the context of word-based information retrieval, our settings are different in several important ways.

- The well studied case of inverted list *intersection* problem is a boolean query: find entries shared by all lists. In our case the query to the index would be to find entries shared by a significant number of the lists but not necessary by all of them.

- The number of words in a natural language is relatively small, compared to the number of possible k-mers we must index.

- Frequencies of different natural language words are vastly different; as a result, average frequency of a word in a language is meaningless. On the other hand, as a genetic sequence has relatively high entropy, the variance of k-mer frequencies is significantly lower.

4.2.2 Inverted Lists Intersection

Approaches to list intersection depend on their representation. Broadly, there are two types of representations: the representations that allow random access to the records and those where the lists are stored compressed and can be decompressed and accessed only in some order, usually sequentially. A simple way to intersect ordered lists is by merging them.

In the case when random access to the lists is available, more efficient algorithms are possible. Given two lists P and Q represented as arrays, if $|P| \ll |Q|$, binary searching all records of P in Q costs $O(|P| \log |Q|)$ and might be faster than merging. Furthermore, assuming that the smallest record of $Q[i]$ not less than $P[i]$ is $Q[i']$, clearly it is enough to search for $P[i+1]$ in $Q[i' \ldots |Q|]$. A family of methods called F-search or *finger search*, introduces various strategies to determine which positions of Q should be tested in order to merge each subsequent record of P. For example, *galloping search* introduced by Bentley and Yao [6], also referred as *exponential search*, tests Q for $P[i+1]$ at locations $Q[i' + 2^k]$, until reaching $Q[i' + 2^k] \geq P[i+1]$ from where it binary searches backwards within the last "hop". Baeza-Yates and Salinger [4] propose searching median of the smaller list in the bigger list. Pairwise merge can be extended to the case of merging

multiple sets, for example, using the *small versus small* (*svc*)approach: run pairwise merge on pairs of the smallest sets recursively, until a single set remains.

Another approach to inverted list intersection is based on hashing [38]. These algorithms generally have two steps. The first step, hashing, uncompresses each of the lists and hashes them to subdivide the problem into smaller subproblems – one for each hash table entry. The second, merge step, solves smaller problems for each of the clusters either recursively or with a hardware-efficient merge implementation.

4.2.3 Fully Indexable Dictionary

Fully Indexable Dictionary is a data structure representing a bit vector B of size u. Two operations must be defined on the representation: for $\Sigma = \{0, 1\}$ and $x \in \Sigma$, $Rank_x(i)$ computes the number of appearances of bit x in $B[1 \dots i]$ and $Select_x(i)$ computes the position of i'th appearance of bit x in B. Most work on this problem revolves around designing space efficient representations allowing for $O(1)$ Rank and Select queries. In this section we overview the major results on this problem and their applicability to inverted indices. As there are $C(u, n)$ bit vectors of length u with n ones, by a combinatorial argument, each of them can be represented using $\log C(u, n) \cong n \log(u/n)$ bits. Yet this counting argument does not prove that we can represent a bit vector in this size and still query it in $O(1)$ time, as allowed with a "conventional" bit vector representation.

The pioneering work of Jacobson [36] introduced the concept of *succinct* data structures representation: a representation of the data structure that takes space close to the information theory lower bound while allowing operations times of the "naive", not memory efficient, representation. Subsequent work on succinct *FID* was focused on achieving the combinatorial bound the bit vector representation and improving the lower order terms, while allowing *Rank* and *Select* in $O(1)$ time. The best result to-date [55] represents a FID in $\log C(u, n) + O(u \log \log u / \log u))$ bits of space.

The main obstacle in using theoretically optimal FID results in practice lies in their lower order terms being practically linear in u, which can be much higher than then the combinatorial bound. Unfortunately, not much can be done about it: Beame and Fich [5] prove that there is no FID representation with constant *Rank* and *Select* times using $\log C(u, n) + n^{O(1)} \log m$ bits. One way around the expensive lower order term problem is by relaxing the time constraints. A common way to measure the complexity of an ordered set is a data-aware measure $gap(S) = \sum_{i>1} \log(S[i] - S[i-1])$. The gap measure counts the space required to encode the distances between successive items. By Jensen's inequality it is never more than $\log C(u, n)$ and is frequently much less. Gupta et al. [31] represent a FID in $gap(S) + O(nloglog(u/n))$ bits and support rank and select in $O(\log n)$ time.

4.3 Read Mapping Algorithm - Outline

In subsequent sections we introduce our read mapping algorithm.

Denote reference DNA sequence with S. Our solution consists of three main stages:

- Construct an index for sequence S.

- Use the index in order to compute the set of *candidate regions* – regions of potential match – for each read.

- Locate matches for each read in corresponding candidate regions using a vector implementation of Smith-Waterman algorithm.

In Section 4.4 we introduce our approach to index a reference genome. Our index construction step is very fast – taking about a minute for a full human genome on a modest PC. As result, unlike with other indexing approaches, our index need not be computed in advance. Building index while mapping makes it possible to search accurately and efficiently for matches in narrow, specific locations on the reference, selected by the user in real time.

As a building block of our index construction we design an algorithm to efficiently transpose bit matrices, described in Section 4.5.

Candidate regions found using the index are relatively long: many times the length of the reads. Next step of the algorithm, discussed in Section 4.6, is to find short windows in candidate regions that are likely to contain a high-scoring match. This step is accomplished with help of an algorithm that finds exact matches between the read and the window.

Finally the algorithm "glues" between regions of exact match between the read and the window using a novel multi-pair vector implementation of Smith-Waterman algorithm, presented in Section 3.2.4, computing many pairs of alignments simultaneously.

4.4 The Indexer: An Application of Hash-based Clustering

4.4.1 Index Definition

In this section we introduce indexing data structure employed by our mapping algorithm.

Denote with $rmax$ an upper bound on the length of the reads. Let us split S into l regions $S_0, \ldots, S_{l-1}$ of equal length S/l, where l is chosen such that $rmax < S/l$ and then extend each region on both sides by $rmax$, i.e. make adjacent regions overlap by $rmax$. Following this construction each read is entirely contained in at least one region and at most two. Furthermore, as there exist 4^k k-mers of length k, if l is chosen such that regions' length $S/l + 2rmax < 4^k$, then the expected number of appearances of a k-mer in a region is $\frac{S/l+2rmax-k+1}{4^k} < 1$.

Given parameters l and k we define index I for DNA sequence S as follows: I is a matrix of 4^k rows of length l bits where $I[i; j] = 1$, for $i \in [0 \ldots 4^k)$ and $j \in [0 \ldots l)$, if and only if region S_j contains k-mer i. For example, if i appears in all regions, $I[i]$ – the i'th row of I – is a vector of l ones. Index I can be constructed using a simple Algorithm 2.

Our index is not limited to simple contiguous seeds and can be easily adopted to be used with spaced seeds (recall 1.3.5) of weight k.

To visualize the index on a concrete example, let S be a string of length $|S| = 4 * 10^9$. Let $l = 8192$ and $k = 11$. Then length of each region is about 500000 (we are assuming

Algorithm 2 Build index I – a boolean matrix of size $4^k \times l$]

Init: $I[i;j] = 0$, for $i < 4^k$ and $j < l$
for $pos = 0$ **to** $pos < |S| - k$ **do**
  $i = pos/l$
  $j = S[pos \dots pos + k - 1]$, i.e. j is k-mer starting at position pos
  $I[i;j] = 1$
end for

that $rmax << 500000$), and the expected number of appearances of a k-mer in a region is about $\frac{500000}{4^{11}} = 0.125$. Memory used by the index is $4^{11} * 8192 \approx 4GB$.

4.4.2 Mapping a Read

Let $R_m = \{r_1, \dots, r_m\}$ be a set of m k-mers in read r. Denote with $B_r = I[r_1] + \dots + I_[r_m]$: a vector of length l containing the bitmap sum of the rows of I corresponding to the k-mers in r.

Consider an alignment of read r versus string S and suppose the alignment is fully contained in region S_i. Then $B_r[i]$ is an upper bound on the number of k-mers in R_m that appear in the alignment. The idea behind our index is to use the values of $B_r[i]$ in order to evaluate the chance that region $S[i]$ contains an optimal match for read r.

Our algorithm establishes a heuristic cutoff τ such that:

1. If there is a match of r in S_i, then $B_r[i] \geq \tau$ with high probability.

2. If $B_r[i] < \tau$, than with high probability r does not match in S_i.

In order to map a set of reads, the algorithm computes column-wise sum B_r as described above for every read r in the dataset and uses the cutoff $B_r[i] \geq \tau$ to find regions S_i corresponding to a possible match.

In addition to the cutoff $B_r[i] \geq \tau$, the algorithm also uses a heuristic threshold $OFFMAX$ discarding regions i s.t. $\max(B_r) - B_r[i] > OFFMAX$, here $\max(B_r)$ denotes the maximum of column-wise sums in B_r.

To map a read to the reverse compliment strand of S the algorithm computes reverse compliment r' of r and uses index I on r'. As the true strand of r is unknown, threshold $OFFMAX$ is applied for both r and r' together, after computing $\max(\max(B_r), \max(B_{r'}))$.

4.4.3 Fast Bitmap Summation

As the datasets used in practice may contain millions of reads, the algorithm usually needs to compute many billions of bitmap summations. A trivial bitmap summation algorithm requires roughly $2lm$ instructions in order to compute B_r: lm instruction to extract the bits and additional lm instructions to sum them up. Next we show how one can utilize the features of a modern CPU in order to do the extraction and the summation in parallel, at the same time improving locality of memory access.

All modern processors have either 8 or 32 bits as the smallest granularity for arithmetic operations. Consequently, there are no built-in operations for bit-wise summation. Hence, in order to find sum of bitmaps it is necessary to "unpack" the bitmaps first, requiring costly sequential operations of bit extraction. To ameliorate this limitation we designed an algorithm that sums up bitmaps on a processor with minimum granularity of 2^k bits, simultaneously with their extraction.

Algorithm 3 Sum_3 computes column-wise sum of bits in bitmaps $B[0 \dots 2]$ of l bits separately for odd and for even positions: compute $S_{odd}, S_{even} = \Sigma_{i<3}(B[i])$

Define: $MASK_k^l = (1^k 0^k)^m$
$S_{odd} = (B[0] \& MASK_1^l) + (B[1] \& MASK_1^l) + (B[2] \& MASK_1^l)$
$S_{even} = (B[0] \& MASK_1^l) >> 1 + (B[1] \& MASK_1^l) >> 1 + (B[2] \& MASK_1^l) >> 1$

Algorithm 3 shows Sum_3 – a vector algorithm to compute the sum of three bitmaps of length l. The algorithm uses a mask to split each bitmaps into two bitmaps of the same length: one for odd positions, another for even. This is done in order to reserve two bits for each position in the sum. Next, Sum_3 computes the sum for the odd and for the even positions separately using regular vector addition instructions. Correctness of Sum_3 follows from the trivial fact that the sum of any three one-digit numbers is at most 3, hence the sum can be represented in 2 bits. If the processor architecture supports vectors of length $vlen$ bits, Sum_3 takes $\Omega(l/vlen)$ instructions compared to $\Omega(l)$ instructions for simple sequential summation.

In the general case, $Sum_{2^{2^{k+1}}-1}$ computes the sum of $2^{2^{k+1}} - 1$ bitmaps where the sum for each position is stored in 2^{k+1} bits. $Sum_{2^{2^{k+1}}-1}$ is computed as follows.

1. Call recursively $\frac{2^{2^{k+1}}-1}{2^{2^k}-1}$ instances of $Sum_{2^{2^k}-1}$, the result for each position is represented in 2^k bits.

2. "Spread out" the results computed in the step above using $Mask_{2^k}^{2^k}$, as in Sum_3, doubling the number of bits reserved for each position to 2^{k+1}.

3. Compute the sum of vectors constructed above using vector addition.

Correctness of $Sum_{2^{2^{k+1}}-1}$ follows from the observation that the sum for each position is at most $2^{2^{k+1}} - 1$ and it can be computed in 2^{k+1} bits without overflow into memory reserved for other positions.

For example, Sum_{15} computes the sum of 15 bitmaps by first invoking five instances of Sum_3, spreading out the results to allow four bits for each position and finally computing the sum using simple add instruction. Recursively, Sum_{255} is computed by "sreading out" and adding up the results of 17 invocation of Sum_{15}.

In our tests on a 128-bit CPU our algorithm achieves more than 30x speedup, compared to the trivial algorithm.

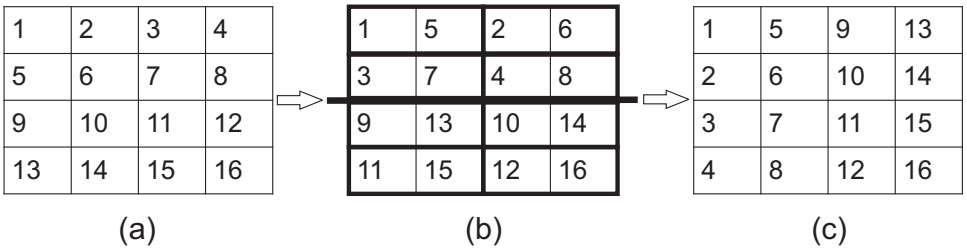

Figure 4.1: Transposing a 4×4 matrix using 2-way merge. **(a)**. Original matrix. **(b)**. Top two rows: merge first two rows of **(a)**, bottom two rows: merge rows 3 and 4 of **(a)**. **(c)**. Merge top and bottom halves of **(b)**, each considered as 4 pairs of elements, i.e. merge {(1, 5), (2, 6), (3, 7), (4, 8)} and {(9, 13), (10, 14), (11, 15), (12, 16)} obtaining {(1, 5), (9, 13), (2, 6), (10, 14), (3, 7), (11, 15), (4, 8), (12, 16)} – a row-major representation of **(a)** transposed.

4.4.4 Index Construction Using In-place Transposition of a Bit-Matrix

Index construction Algorithm 2 initializes index I in a column-by-column order. Consequently, if I were stored in row-major order, the performance would be poor due to lack of locality of memory accesses. If the matrix were stored in a column-major order, than the summation of rows of I would be inefficient for similar reason.

Hence it would be beneficial to construct I in column-major order and transpose it afterwards. In Section 4.5 we present an efficient vector algorithm for bit-matrix transposition.

4.5 Transposition of a Bit-Matrix

4.5.1 Overview

In this section we consider a matrix transposition problem: given a bit matrix I transform it into a matrix I' s.t. $I'[i; j] = I[j; i]$. All matrices are represented in row-major order. The outline of the section is as follows. First, we provide a brief history of the problem. Next we develop a transposition algorithm for a square matrix with the number of rows and columns a power of 2. Subsequently we discuss how to apply the algorithm to our problem: transposing a rectangular matrix of index I.

While the problem of I/O efficient matrix transposition is well-studied, the author is unaware of efficient algorithms to transpose a bit-matrix on a vector processor. Pioneering work on the subject of I/O efficient matrix transposition by Floyd [26] proves matching upper and lower bounds of $O(\frac{N}{B \log N})$ to transpose a matrix of size $m \times n$, here $N = mn$, on a machine with memory size M and cache line length B. Floyd's lower bound applies for a restricted case of $B = \Theta(M) = \Theta(N)^c$ for $0 < c < 1$. Aggarwal and Vitter [2] extend the result of Floyd proving matching lower and upper bounds for matrix transposition depending on the values of memory model parameters B and M. Aggarwal and Vitter [2] show external memory matrix transposition algorithm using a B-way merge. Figure 4.1 illustrates the algorithm for $B = 2$. This algorithm cannot be adapted efficiently for

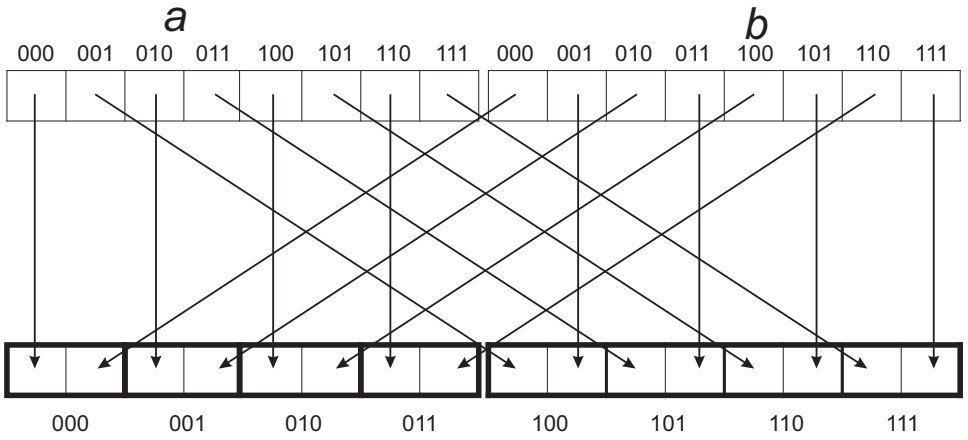

Figure 4.2: *Shuffle-swap* of sequences a and b.

vector processing as merging changes distances between blocks of data.

Frigo et al. [27] prove matching lower and upper bound for the matrix transposition problem in cache-oblivious memory model. Assuming a *tall cache* memory model – a cache of size $\Omega(B^2)$, where B is a cache line length – they show a $O(\frac{mn}{B})$ cache misses algorithm and prove that it is optimal.

4.5.2 Transposition of a Square Bit-Matrix

Our algorithm employs logical, instead of arithmetic operations on data, hence types of each record $I[i;j]$ is not restricted to be an integer type, instead it can be a single bit-wide, as required for our indexing application. The idea of the algorithm is to represent matrix transposition as a combination of a small number of data permutations that can be vectorized: for matrix I of size $2^m \times 2^m$ the algorithm requires m parallel vector operations. In our description I denotes both the matrix and its row-major representation of length 2^{2m}. The notation would be clear from the context in which it is used.

First, let us introduce some notation. For m-bit number $i = \overline{i_{m-1}i_{m-2}\cdots i_0}$ define $i^{\circlearrowleft} = \overline{i_{m-2}\cdots i_0 i_{m-1}}$. Given two sequences a and b of 2^m records, *shuffle-swap* of a and b, is a sequence c of 2^m *blocks* of pairs of records: $c[i] = a[i^{\circlearrowleft}]b[i^{\circlearrowleft}]$.

The construction is illustrated in Fig. 4.2. One can see that unlike a simple merge, *shuffle-swap* does not change the distance between blocks of data it moves. This makes it possible to implement *shuffle-swap* using a few highly regular vector instructions:

1. Shift b to the right to align swapped blocks
2. Apply *XOR-swap* [68] algorithm with a bit-mask set at positions to be swapped.
3. Shift b back to the left.

The algorithm is illustrated in Fig. 4.3. It is executed machine vector-width at a time, in a single scan through a and b.

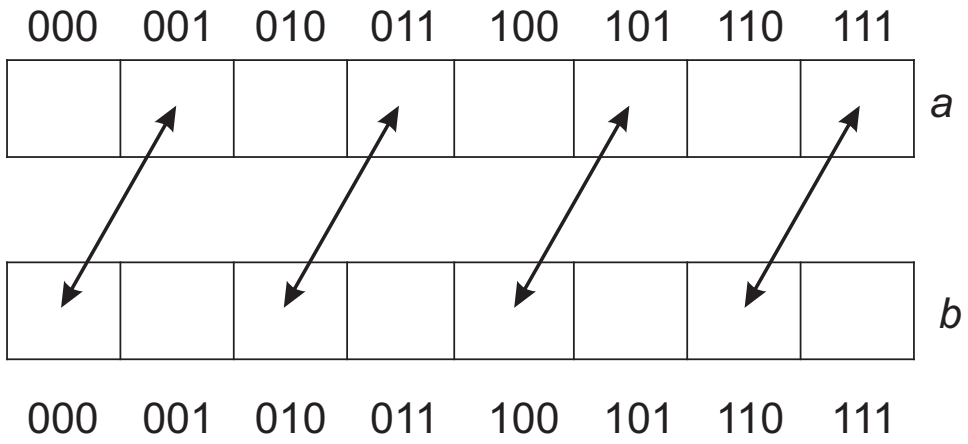

Figure 4.3: Computing swap-shuffle of a and b in-place, using shifts and swaps of equidistant memory blocks. Vector swaps of memory blocks are computed using a vector version of XOR SWAP algorithm[68].

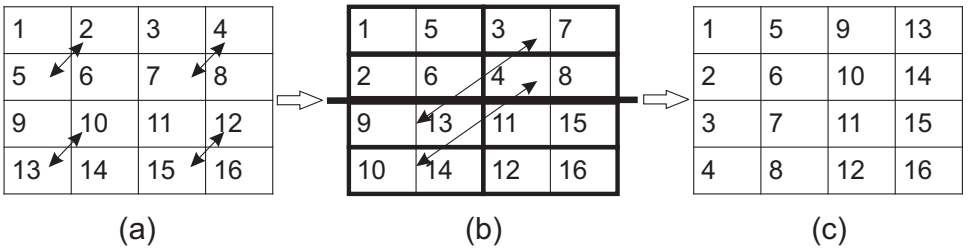

Figure 4.4: Transposing a 4×4 matrix using two levels of swap-shuffles. **(a)**. Original matrix. The arrows show records to be swapped in the first level. **(b)**. After the first level of swap-shuffles. Arrows show blocks of two records to be swapped in the second level. **(c)** The result of the second level of swap-shuffles.}

Before proceeding to the description of our transposition algorithm, note two simple properties of c that follow directly from its definition:

- In each record $c[i]$ a record of a precedes a record of b.
- The destination of records $a[i]$ and $b[i]$ are in block $i' = i^{\circlearrowleft}$ of c.

The algorithm to transpose I is a composition of m levels of shuffle-swap steps. In the first level it performs 2^{m-1} *shuffle-swaps* on pairs of consecutive rows – $2i$ and $2i+1$ – of I, building 2^{m-1} vectors of blocks of 2 elements. Each subsequent level computes *shuffle-swaps* on pairs of consecutive vectors of blocks constructed in the previous level. The algorithm is illustrated in Fig. 4.4.

Lemma 1 *Matrix I' computed by m levels of* shuffle-swaps *is I transposed. The complexity of the algorithm is $O(N \log N)$ and the number of cache misses is $O(\frac{N \log N}{B})$, where N is the number of records in the matrix.*

Proof: The result of level i of *shuffle-swaps* is a representation of 2^{m-i} vectors of blocks with 2^m blocks in each vector. As with the merge-based algorithm of[2], each such block

contains a group of 2^i records that appear consecutively in some row of I-transposed. Hence, after m levels, the algorithm constructs a single vector I' of 2^m blocks, where each block is one of the rows of I transposed.

It remains to show that the rows of I' follow in the correct order. Let r be an arbitrary row of I and let us see where an arbitrary $r[i]$ ends in its corresponding vector following $k \leq m$ levels of permutations by *shuffle-swaps*. It is easy to see by induction on k that its destination lies in block $i^{\circlearrowright k}$, i.e. an m-bit i circular shifted right by k bits. For $k = m$, the circular shift is an identity function, hence the destination of $r[i]$ for any row r of I lies in block i of I'. Since each block of I' is some row of I-transposed, we proved that I' is a row-major representation of I-transposed.

Our in-place algorithm performs $\log N$ sequential scans of $O(N)$ data entries each. Hence it makes $O(N \log N/B)$ page faults.

Q.E.D.

While this transposition algorithm is suboptimal in both time complexity, by a factor of $\log N$, and in the number of cache misses in the standard memory model [2], it is likely to benefit from pre-fetching, not captured by the standard model, thanks to sequential memory access [66]. Practical comparison to the optimal matrix transposition algorithms was beyond the scope of this work as we were only interested in efficient transposition of bit-matrices.

4.5.3 Application to Non Square Matrices

Instead of extending our transposition algorithm to general matrices, we consider a special case of the problem relevant to our index construction: transpose a bit matrix I of size $2^k \times 2^m$ constructed row-by-row, where writes within each row are in random order.

As discussed earlier, after applying k levels of $shuffle-swaps$ on rows of I we obtain a single vector V containing 2^m blocks of 2^k records in each – each block contains one of the columns of I transposed. For an arbitrary row r of I record $r[i]$ ends in block $i^{\circlearrowright k}$ of V.

Hence, if each row of I is reordered by sorting the indices with transformation $\circlearrowright (m-k)$, then after applying k levels of $shuffle - swaps$ on the rows of the pre-processed matrix we obtain I transposed. In our case we just fill each row of matrix I after applying $\circlearrowright (m-k)$ transformation on records' indices. As each row is filled in random order, this transformation does not introduce an additional performance hit.

4.6 Post-Indexing

4.6.1 Computing Exact Matches Between Reads and Sliding Candidate Windows

Hashing step of section 4.4 computes for each read r a set of *candidate regions* S_j: regions containing at least τ k-mers of r. Using sorting our algorithm collects for each region S_j a set of reads for which S_j is a candidate region. We call this set a set of *candidate reads*

for region S_j. In order to achieve efficient vectorization, our algorithm aligns in parallel all reads in the candidate set versus corresponding candidate region.

The post-indexing stage for candidate reads for region S_j is comprised of two parts. The first step is to find a set of *candidate windows* – substrings of S_j of length close to $|r|$ such that there exists a set of τ k-mers r_1, ..., r_τ of r that appear in both r and the candidate window in the same order. Each candidate window is represented by a set of exact matches in relatively close proximity of one another. In current implementation the post-indexing stage is accomplished using a simple algorithm shown below:

1. Init: $i = 0; k = 0$
2. Prepare vector P of tuples $(p_1; p_2)$ such that a k-mer in S_j at position p_1 is the same as a k-mer in r at position p_2
3. Sort P, primary in increasing order of p_1 and secondary in decreasing order of p_2.
4. Advance i and k, finding windows $P[i \dots i+k]$ such that $P[i+k].p_1 - P[i].p_1 \approx |r|$. While "sliding" i and k, maintain the count Inc of the number of position i in the current sliding window s.t. $P[i].p_2 < P[i+1].p_2$
5. If $Inc > \tau$, compute Longest Increasing Subsequence LIS for $pos_r(P[i \dots i+k])$ step above.
6. If two k-mer matches at positions $LIS[j]$ and $LIS[j+t]$ are not adjacent, i.e. they are not a part of the same exact match of length $k+1$, add a pair of strings between them to the "glue set" SW_SET - a set of alignments we compute later using a vector Smith-Waterman algorithm.

The complexity of the algorithm shown above depends on the number of repetitions of matching k-mers. In the worst case, the size of P is a product of lengths of S_j and r. Computing a Longest Increasing Subsequence of n numbers is done in $O(n \log n)$ time. Counter Inc maintained by the algorithm provides an upper bound on the length of LIS and is used in order to reduce the number of "sliding" windows $P[i \dots i+k]$ for which LIS needs to be computed. Another heuristic reduces the number of Smith-Waterman alignments the algorithm has to compute: after computing the LIS we estimate the best possible score for the window, compare it with the current best score for the read, dismissing the window if estimated score is too low.

The post-indexing algorithm is run on the set of all candidate reads for region S_j, computing a set SW_SET of alignment problems required to "glue" corresponding exact matches. The next step is computing all alignments in SW_SET using a variant of Smith-Waterman (SW) algorithm with affine costs corresponding to the error model of the reads. In our work we compute SW matrices for multiple pairwise alignments simultaneously. This way we are able to fully utilize vector processing unit by altogether avoiding data dependencies between different records of the vector.

For a typical dataset SW_SET contains thousands of alignment problems. To simplify our implementation we first cluster SW_SET into sets of problems of identical size. Henceforth, we assume that all problems in SW_SET are of the same size.

4.6.2 Vectorized Smith-Waterman to Compute Multiple Pairwise Alignments of Genetic Sequences

Notation

In this section we describe a vector algorithm that given a set of l pairs of DNA or protein sequences, computes l Smith-Waterman alignments in parallel. The algorithm for DNA sequences is described first, then we show an extension for vector alignment of pairs of protein sequences: as we shall see, larger domain size and hardware limitations prevent a straightforward adaptation of the DNA algorithm for proteins.

First, we introduce some notation:

- $\mathcal{L}$ and $\mathcal{R}$ — two sets of size l containing sequences of length m and n, respectively.
- $\mathcal{L}[i]$ or $\mathcal{R}[i]$ — sequence i in set $\mathcal{L}$ or $\mathcal{R}$, respectively.
- L_i — a vector of records at positions i in the sequences of $\mathcal{L}$: $L_i[k] = (\mathcal{L}[k])[i]$. R_i is defined similarly.
- $r_1 \circ r_2$ — substitution score between a pair of amino acids or nucleotides r_1 and r_2. The scores are defined by the domain.
- $Q = L \circ R$ — for two vectors L and R of size l vector Q of size l is a substitution vector between them, i.e. $Q[i] = L[i] \circ R[i]$. If all records of R are identical, i.e. $R[i] = r$ for all i, we use a notation $Q = L \circ r$.

Scalar Smith-Waterman Algorithm

Let us briefly remind a scalar version of Smith-Waterman algorithm. Smith-Waterman algorithm consists of two stages, the *forward* and the *backtracking* [62]. For two strings s_1 and s_2 of lengths m and n respectively the forward stage computes optimal partial alignments for all prefixes of strings. The partial alignments are represented in matrix M of size $m \times n$, where $M[i; j]$ stores the scores of partial alignment of $s_1[1 \dots i]$ and $s_2[1 \dots j]$ as well as the backtracking information in order to traverse the best alignment. This matrix is computed in non-decreasing order of column and row indices and record $M[i; j]$ depends on the substitution score of $s_1[i]$ and $s_2[j]$ and the values in M computed earlier.

After the forward stage completes, the backtracking stage uses matrix M computed in the forward stage in order to output an optimal alignment [1]. The complexity of the forward stage of the scalar algorithm is $O(mn)$; the complexity of the backtracking stage is $O(m + n)$.

[1] As a practical note, matrix M of partial alignment scores contains more data than required for backtracking: it is possible to store only records of M that are required for further dynamic programming computation, such as a single row or a column, while keeping data required for backtracking in just 4 bits per each position.

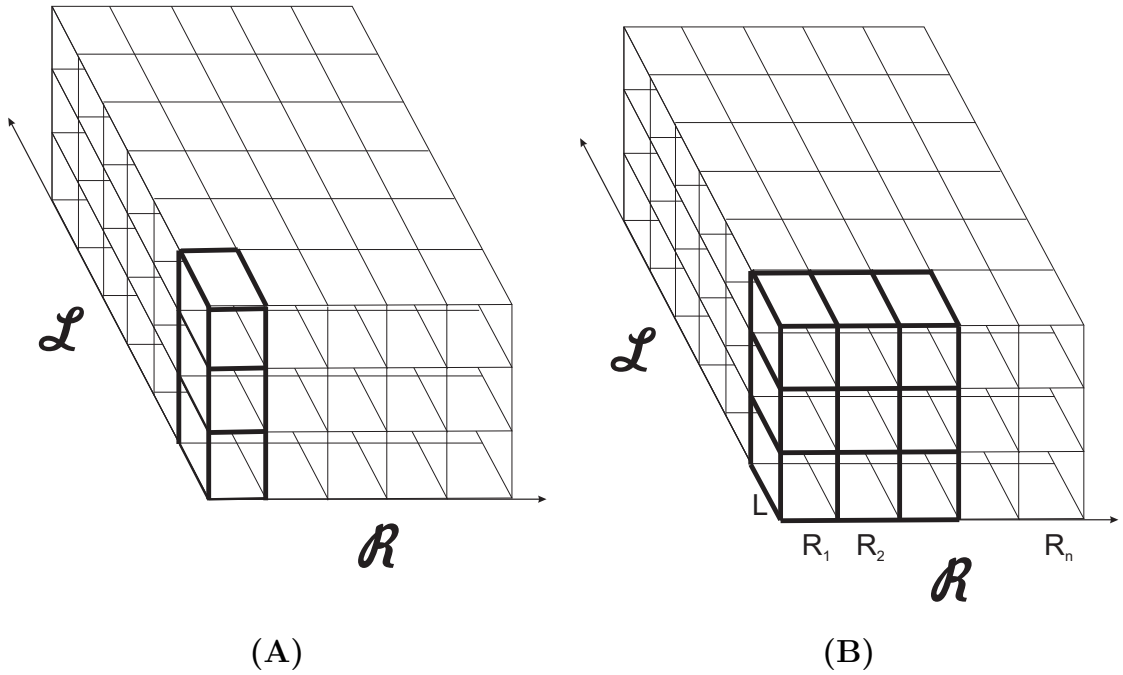

Figure 4.5: Computing pairwise alignment of two sets of sequences $\mathcal{L}$ and $\mathcal{R}$. Each set contains l sequences. The $3D$ matrix has l "layers", one layer for each pair of sequences. In the figure $l = 3$. **(A)** For DNA sequences the algorithm computes one *stack* (shown bold) of the $3D$ matrix at a time using vector instructions. The computation for each stack $(i; j)$ depends on the stacks of the matrix computed earlier and the vector of substitution scores: $L_i \circ R_j$. **(B)** For sequences of amino acids the substitution scores are computed for l stacks in advance using matrix transposition algorithm.

Vectorized Smith-Waterman – Computing Multiple Alignments Simultaneously

In this section we describe the algorithm in general terms. Domain-specific details are deferred to later sections.

Both for DNA sequences and for proteins, our vector Smith-Waterman algorithm allocates a matrix $M[m \times n]$ of vectors of l records; we call these vectors *stacks of* M. Matrix M represents l matrices of size $m \times n$, called *layers*, where layer i is a Smith-Waterman matrix for the pair $\mathcal{L}[i]$ and $\mathcal{R}[i]$. See Fig. 4.5.

The key observation we use in our algorithm is that the the computation of each stack of M depends only on the stacks of M computed earlier and the vector of substitution scores corresponding to the current stack of M. If the vector of substitution scores is available, a vector computation of a column can be done by a straightforward translation of the instructions of the scalar Smith-Waterman algorithm to their vector analogs.

As the backtracking paths are different for different pairs of strings, the alignment is produced separately for each of the l problems using corresponding layer of M. Yet, time complexity to output each alignment is linear and the algorithm's complexity is dominated by the computation of M.

Computing Substitution Scores for DNA sequences

In our solution to compute $L \circ R$, where L and R are vectors of nucleotides, we use a variant of *Shuffle* instruction we introduced in Section 1.3.6. In the following description $l = 16$, consistent with Intel's implementation of the instruction.

Assuming that nucleotides in the sequences are represented with integers 0 to 3, we define translation E as an alternative representation for the nucleotides: $E(0, 1, 2, 3) = 1, 2, 4, 8$. Define function $ScoreIdx$ as

$$ScoreIdx(a, b) = ((((a|b) >> 1) \wedge (a|b))\&0b1111) - 1$$

where $|$, $\wedge$ and & stand for bitwise OR, XOR and AND, respectively. It is trivial to observe that $ScoreIdx$ is symmetric and for ordered pairs $\{l_1, r_1\}$ and $\{l_2, r_2\}$ s.t $0 \leq l_i \leq r_i \leq 3$ it holds that:

$$ScoreIdx(E[l_1], E[r_1]) = ScoreIdx(E[l_2], E[r_2]) \Leftrightarrow (l_1, r_1) = (l_2, r_2)$$

and

$$0 \leq ScoreIdx(E[l_i], E[r_i]) \leq 15$$

It follows from the properties of $ScoreIdx$ that

$$L \circ R = Shuffle(Scores, ScoreIdx(E[L], E[R]))$$

where a 16-element vector $Scores$ is defined as

$$Scores[ScoreIdx(E[l], E[r])] = S[l; r]$$

It is easy to see that if we define $E[5] = 16$ the properties of $ScoreIdx$ shown above continue to hold. This extends the construction to domains of size 5 with 15 substitution pairs. The method is limited to at most 16 substitution pairs and is not applicable to domains of size higher than 5, such as the domain of amino acids.

4.6.3 Vector Smith-Waterman to Compute Multiple Pairwise Alignments of Protein Sequences

In this section we briefly discuss an adaptation of our vector Smith-Waterman algorithm to the problem of aligning a set of proteins to a reference protein sequence.

The key difference between the two problems is that there is no clear way to adapt the algorithm from Section 4.6.2 to compute substitution scores $Q = L \circ R$ for two vectors of $l = 16$ amino acids as it would require computing a shuffle on a vector of at least 210 records – the number of substitution pairs of 20 main amino acids, while only 16 elements can be shuffled using a hardware instruction. Due

It is not possible to use SWIPE – a vector Smith-Waterman implementation by Rognes [58] – for our problem. SWIPE can only align a set of sequences to a *single query sequence*. The single query sequence limitation is at the core of SWIPE's algorithm to compute substitution scores $Q = L \circ r$: given a scoring matrix M representing

substitution scores for all pairs of amino acids, SWIPE first extracts a single row $M[r]$ containing all substitutions for amino acid r. Then $L \circ r$ can be computed simply as $L \circ r = Shuffle(M[r], L)$. To be more precise, as there are 20 amino acids, $L \circ r$ is computed using two Shuffle instructions on 16 elements.

Instead of computing a single substitution vector $Q[i; j] = L_i \circ R_j$ we can compute in parallel matrix I of size $l \times l$ containing l substitution vectors $Q[i; j], Q[i; j+1], \ldots Q[i; j+k-1]$ required to compute next l stacks of M. Using I, the algorithm computes corresponding l stacks of M as discussed in Section refsubst:gen, before proceeding to the next l stacks. The construction is illustrated in Fig. 4.5.(B).

While computing I in "stack-major" order is difficult as it requires a shuffle of a 210 element matrix, each of its rows can be computed using two Shuffle instructions, as explained above. Hence, it is possible to compute I using a two step vector algorithm:

- Compute a row-major representation of I using Shuffle instructions.
- Transpose I using vector transposition algorithm from Section 4.5.

4.7 Results

To test our aligner we simulated 800000 reads of different lengths and error rates using reference sequence human_g1k_v37 of human genome. We compared our results to those of the most recent variant of BWA – BWA MEM [42], one of the best performing and popular mapping programs. The error rates were 2%, 5% and 10% and substitution errors were introduced 8 times more likely than either insertion or deletion errors. Insertion/deletion length l was distributed as geometric distribution $P(l) = p(1-p)^l$ with $p = 0.7$.

The tests were run on a single core of Lenovo w520 laptop with core i7 2720qm CPU and 16GB of RAM. Our index was built with a spaced seed 111010010100110111 of weight 11 and length 18, the seed used by PatternHunter [45]. A simple seed of length $k = 11$ was used in the post-indexing stage. Index bit matrix was of size $4^{11} \times 20480$, i.e. about 10GB. The chaining in the post-indexing stage was executed with k-mers for $k = 11$.

The results are summarized in Table 4.1. In our tests, among the alignments our algorithm tests, it outputs only those with the single highest score. If the original position for the read was not among them, the read was considered unmatched. The read does not match for either of two reasons. First, the position was among the candidate windows but it did not result in the highest scoring match: due to errors introduced while generating the read another position scored higher. Second, the position was not among the candidates tested: either the true candidate region for the read was discarded by the indexer, or the position in the true candidate region was skipped in the post-indexing stage. In Table 4.1 a read is considered "matched" by the indexer if the correct position was among the candidate regions for the read that were passed to the post-indexing stage.

Depending on the cutoff thresholds for the indexer, each read can have zero or more candidate regions passed to the post-indexing stage. While more candidate regions result in higher recall of the aligner, at the same time this increases the runtime of the

post-indexing stage. Total number of candidates over all reads in our tests is shown in Table 4.2.

In the testing we found that our mapping algorithm performs comparably to BWA MEM. In addition, if full index cannot be hold in memory, the algorithm splits the reference into parts of size sufficiently small for their index to fit in memory. Unlike suffix array or tree based approaches, splitting of the reference does not incur performance hit for our indexer.

Read Length	125			250			500		
Algorithm	Index	Full	BWA	Index	Full	BWA	Index	Full	BWA
2%: no hits	5072	**6840**	11397	652	**1778**	5069	58	**1139**	2493
5%: no hits	12473	**16248**	17189	2343	**4615**	6188	552	**2651**	2953
10%: no hits	37530	**47209**	66167	9263	14297	**13856**	1094	5738	**4862**
2%: runtime	402	709	**319**	628	1119	**687**	1094	2507	**1410**
5%: runtime	398	802	**575**	627	1281	**1192**	1092	2504	**2461**
10%: runtime	404	1107	**712**	629	**1613**	1852	1089	**2587**	3858

Table 4.1: Comparison of our aligner to BWA MEM for different read length and error rates. Column *Index* shows the results of the indexer stage separately. Columns *Full* and *BWA* show the results of our full aligner and BWA MEM respectively. Row *runtime contains the runtime of each test; row* Mapped shows the number of reads mapped correctly. Index construction time was included in the runtimes for the columns *Index* and *Full*. The best result is bolded.

Error rate	125	250	500
2%	2337458	1336675	1033138
5%	3229136	1985371	1258769
10%	8908922	4352337	2283748

Table 4.2: The total number of (region, read) pairs passed for verification by the post-indexing stage for a set of 800000 reads of different lengths and different error rates (out of the total 20480 regions for each direction of the reference).

4.8 Discussion

In this chapter we introduce a novel algorithm for alignment of long high error rate reads versus reference genome. An important feature of our method is real time index construction that allows efficient and accurate mapping to specified regions of interest. Our algorithm is highly efficient: despite not pre-computing the index, its run time is comparable to that of the best tools that use an index prepared ahead of time.

Among the different parts of the aligner our main focus was on efficient index filtering and on the novel multi-pair Smith-Waterman algorithm. At the same time, there are more

efficient ways to compute exact matches than a simple k-mer chaining method presented in Section 4.6.1 which might greatly improve the performance of the post-indexing stage.

Due to highly regular linear memory access pattern most of our algorithms will readily benefit from advances in hardware, such as doubling of vector length and of cache bandwidth in new Intel Haswell processors [61]. The speed-up for more complex suffix array/tree based methods would likely be more limited. Our algorithms are also much easier to translate efficiently to other vector architectures such as GPUs.

We believe that the algorithms we presented as building blocks for our aligner can be useful beyond its scope. For example, our vector multi-pair Smith-Waterman can be used as a direct replacement for single alignment implementations of Smith-Waterman in other computational biology tools. An interesting question for further study is a hybrid mapping approach where our indexer is used to quickly reduce the size of the problem followed by an application of a suffix array/tree based method in order to map reads to the much shorter candidate regions.

Chapter 5

Approximation Algorithm for Coloring of Dotted Interval Graphs

5.1 The Problem

In this chapter we provide a simple greedy algorithm for coloring of dotted interval graphs with approximation ratio of $\frac{2D+4}{3}$. In order to prove the complexity bound we simplify and generalize the upper bound on the number of non-attacking queens on a triangular board considered by Nivasch and Lev [54].

5.2 Definitions and Notations

Basic definitions:

- $\sharp S$ — the number of elements of set S
- $I(j_0, d, k) = \{j_0, j_0 + d, \ldots, j_0 + dk\}$ for j_0, d and k positive integers — a d-periodic dotted interval.
- D — maximum period d of any dotted interval $I(j_0, d, k)$.
- $n(j, d) = \sharp\{I(j_0, d, k) | j \in I(j_0, d, k)\}$ — the number of d-periodic dotted intervals containing j.
- $n(j) = \sum_d n(j, d)$ — the number of dotted intervals containing j. By definition of coloring, all intervals sharing node j must be colored in different colors.
- $N = \max_j n(j)$. This is obviously a lower bound on the number of colors used by any coloring.
- m — the maximum value of any point (dot) belonging to any interval.

The following notations are used throughout the description of the current state of the algorithm. For dot i and period d let us denote:

- C_i^d — the set of colors used for coloring of the d-periodic intervals having endpoints at both sides of $i + 0.5$. $\sharp C_i^d = c_i^d$.
- $C_i = \bigcup_d C_i^d$ — the set of colors used for the intervals having endpoints at both sides of $i + 0.5$. $\sharp C_i = c_i$.
- $\overline{C} = \bigcup_i C_i$ — the set of colors used by the algorithm. $\sharp\overline{C} = \overline{c}$.
- X_i^d — the set of colors used for coloring of d-periodic intervals ending at i. $\sharp X_i^d = x_i^d$
- Y_i^d — the set of colors used for coloring of d-periodic intervals passing through (containing but neither ending nor starting at) i. $\sharp Y_i^d = y_i^d$
- Z_i^d — the set of colors used for coloring of d-periodic intervals going above i, i.e. having endpoints at both sides of i but not containing i. $\sharp Z_i^d = z_i^d$. Note that if $z_i^d > 0$, then $d \geq 2$.
- P_i^d — the set of d-periodic intervals starting at i, i.e. the set of all $I(i, d, *)$. $\sharp P_i^d = p_i^d$.
- T_i^d — the set of colors used for coloring of d-periodic intervals starting at i. $\sharp T_i^d = t_i^d$. Note that $t_i^d = p_i^d$.
- X_i, Y_i, Z_i, T_i and P_i are defined similarly to C_i — as unions of the corresponding sets over all possible periods d.

  Note that $X_i \bigcup Y_i \bigcup Z_i = C_{i-1}$.

5.3 The Algorithm

The algorithm colors the graph from left to right coloring all intervals starting at i before continuing to $i + 1$. In order to color the intervals of P_i^d the algorithm first reuses the colors of set Re_i^d — the colors of intervals of Z_i^d that are not used to color d-periodic intervals ending at or passing through i. If this was not enough to color all of P_i^d, the algorithm reuses colors from set Re_i — the colors already used and released, i.e. not used by any interval passing above $i - \frac{1}{2}$. Otherwise it allocates a new color. In the pseudocode below $color(v) = pop(S)$ stands for assigning to node v of any color from set S and removing of the just assigned color from S.

$C = \emptyset$
for $i = 1$ to m **do**
  $Re_i = C \setminus C_{i-1}$
  for $d = 1$ to D **do**
    $Re_i^d = Z_i^d \setminus (X_i^d \bigcup Y_i^d)$
    for all p such that $p \in P_i^d$ **do**
      if $Re_i^d \neq \emptyset$ **then**
        color(p)=pop(Re_i^d)

```
        else if Re_i ≠ ∅ then
          color(p)=pop(Re_i)
        else
          c=new Color
          C = C ⋃ {c}
          color(p)=c
        end if
      end for
    end for
  end for
```

5.4 Basic Analysis of the Algorithm

Lemma 2 *For all i and d it holds that $c_i^d \leq N$.*

Proof: The proof is by induction on i and for fixed d.

- For $i = 1$ obviously c_i^d is the number of d-periodic intervals starting at 1 and the claim is trivial.
- Suppose that the claim holds for $i-1$, that is $c_{i-1}^d = x_i^d + y_i^d + z_i^d \leq N$. Let us prove that $c_i^d = y_i^d + z_i^d + t_i^d \leq N$. Two cases are possible:
  - The colors used for coloring intervals of P_i^d were reused from Re_i^d, i.e. $T_i^d \subseteq Re_i^d = Z_i^d \backslash (X_i^d \bigcup Y_i^d)$. Then by inductive assumption $c_i^d \leq c_{i-1}^d \leq N$
  - After reusing Re_i^d the algorithm had to use colors from Re_i or allocate new colors. In this case C_i^d is contained in the set of colors used for coloring of d-periodic intervals containing i. Hence by definition of N $c_i^d \leq n(i,d) \leq N$

Q.E.D.

Lemma 3 *For the number of colors $\overline{c}$ used by the algorithm it holds that*

$$\overline{c} \leq \max_i c_i + N$$

Proof: Let us prove by induction on j that

$$\sharp \bigcup_{1 \leq i \leq j} C_i \leq \max_i c_i + N$$

- Basis: for $j = 1$ the claim is trivial.
- Suppose that the claim holds for $j-1$. That is

$$\sharp \bigcup_{1 \leq i \leq j-1} C_i \leq \max_i c_i + N$$

Two cases are possible:

– $Re_j \subseteq T_j$, i.e. all colors of Re_j were used for coloring of P_j. In this case

$$\bigcup_{1\le i\le j} C_i = \bigcup_{1\le i\le j-1} C_i \bigcup T_j = C_{j-1} \bigcup T_j \ ,$$

consequently $\sharp \bigcup_{1\le i\le j} C_i \le \sharp C_{j-1} + \sharp T_j \le \sharp C_{j-1} + N$ and the claim of the lemma follows.

– $T_j \subseteq Re_j \bigcup C_{j-1}$. In this case no new colors are allocated when the algorithm executes for dot j, hence

$$\bigcup_{1\le i\le j} C_i = \bigcup_{1\le i\le j-1} C_i$$

and the claim of the lemma holds by the inductive assumption.

Q.E.D.

5.5 Bound on C_i and the Multi-Queens on a Triangle

For any $k > 0$ let us enumerate $2D$ possible endpoints of the intervals of period at most D passing over $y = k + 0.5$ accordingly to their distance from y: counting from 0, we label them as follows:

$$D-1, \ldots, 1, 0(y)0, 1, \ldots, D-1$$

To resolve the ambiguity in the labeling, the vertices to the left (respectively, right) of y will be called *left* (respectively, *right*) vertices.

For every color in C_k let us choose a single representative dotted interval colored with this color. Of this dotted interval let us consider a single segment (interval) passing over y. We obtained a bipartite graph G' with $2D$ vertices and c_k edges. We shall represent this graph, which might contain multiple edges, with a $D \times D$ matrix M where $M(i,j) = n$, n being the number of edges $e = (i,j)$ from the left vertex named i to the right vertex named j. For every (i,j) s.t. $M(i,j) \neq 0$, the length of the corresponding interval is $i+j+1$, hence $i+j \le D-1$. Consequently, assuming that $(0,0)$ corresponds to the upper-left corner, all entries of G below the *main antidiagonal*, defined as $\{(i,j)|i+j = D-1\}$, are zero. We call such a matrix an *upper triangular matrix* of size D.

The following observations will be used in the subsequent discussion:

- The sum of entries in row i of this matrix corresponds to the number of segments having one endpoint at left vertex i and another to the right of y. Since the degree of each vertex is at most N, this number is bounded from above by N.

- Similarly, the sum of entries in column j is the size of a subset of edges incident on right vertex j and is bounded from above by N.

- The sum of entries on *antidiagonal* $\{(i,j)|i+j = d-1\}$ equals to the number of differently colored intervals of length d passing over $k+0.5$, i.e. c_k^d. By Lemma 2 this sum is also bounded from above by N.

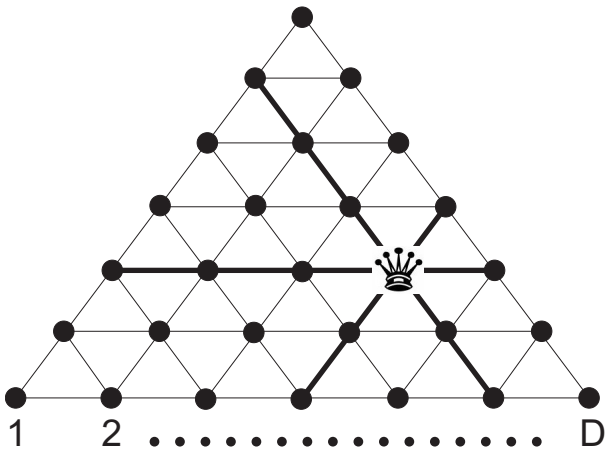

Figure 5.1: Attacks of a queen. Bold lines pass through the attacked nodes. Each queen attacks $2D+1$ tiles — we consider a tile occupied by a queen to be attacked by it three times.

Lemma 4 *For an upper triangular non-negative integer matrix M=D × D, with the sum of entries on each row, column and top-to-left diagonal (antidiagonal) bounded from above by N denote by $Q(D,N)$ the maximum possible sum of entries of the matrix. Then $Q(D,N) \leq N(2D+1)/3$.*

Let us rephrase Lemma 4 in a more symmetric way using chess notation, in terms of what we call a *multi-queen placement problem on a triangle*. We say that a node t_1 is attacked by a queen on node t_2 if both t_1 and t_2 are on some line parallel to any of the three sides of the triangle, see Figure 5.1.

Lemma 5 *Given a board in the shape of equilateral triangle of size D, see Fig. 5.1. Denote by $Q(D,N)$ the maximum number of queens that can be placed on the board such that for any row parallel to any of the three sides there are at most N queens on the row; we allow more than one queen placed on a node. Then $Q(D,N) \leq N(2D+1)/3$.*

Proof: As in Nivasch and Lev [54] our argument uses counting of the number of attacks mounted by the queens, denote it with *Att*, on the board's nodes, though our analysis is more symmetric.

Denote $q = \frac{Q(D,N)}{N}$.

It can be easily seen from Fig. 5.1 that every queen, independently of placement, mounts exactly $2D+1$ attacks — we consider a node occupied by a queen to be attacked by it three times. Hence qN queens mount exactly

$$Att = qN(2D+1) \tag{5.1}$$

attacks.

Let us look at the number of attacks that qN queens can mount from another point of view. Consider first only the way to maximize the number of horizontal attacks. Let us place the first N queens on the row of length D contributing ND horizontal attacks.

Next N queens are placed on the row of length $D-1$ and contribute $(D-1)N$ attacks. This process is continued for $\lfloor q \rfloor$ rows utilizing $\lfloor q \rfloor N$ queens. The remaining $(q - \lfloor q \rfloor)N$ queens are placed on row of length $D - \lfloor q \rfloor$. Clearly this way we get an upper bound on the number of horizontal attacks mounted by any placement of qN queens. Denote it by H:

$$H = \sum_{i=0}^{\lfloor q \rfloor - 1} (D-i)N + (D - \lfloor q \rfloor)(q - \lfloor q \rfloor)N = \frac{(2D - \lfloor q \rfloor + 1)\lfloor q \rfloor N}{2} + (D - \lfloor q \rfloor)(q - \lfloor q \rfloor)N \tag{5.2}$$

It is easy to see that:

$$H = \frac{(2D - \lfloor q \rfloor + 1)\lfloor q \rfloor N}{2} + (D - \lfloor q \rfloor)(q - \lfloor q \rfloor)N \leq \frac{(2D - q + 1)qN}{2} \tag{5.3}$$

Indeed, after rearranging and dividing both sides by $N/2$, using notation $\{q\} = q - \lfloor q \rfloor$ we have:

$$\begin{aligned}
&(2D - q + 1)q - (2D - \lfloor q \rfloor + 1)\lfloor q \rfloor - 2(D - \lfloor q \rfloor)(q - \lfloor q \rfloor) &= \\
&2Dq - q^2 + q - 2D\lfloor q \rfloor + \lfloor q \rfloor^2 - \lfloor q \rfloor - 2Dq + 2D\lfloor q \rfloor + 2\lfloor q \rfloor q - 2\lfloor q \rfloor^2 &= \\
&- q^2 + q + 2\lfloor q \rfloor q - \lfloor q \rfloor^2 - \lfloor q \rfloor &= \\
&- (\lfloor q \rfloor + \{q\})^2 + (\lfloor q \rfloor + \{q\}) + 2\lfloor q \rfloor(\lfloor q \rfloor + \{q\}) - \lfloor q \rfloor^2 - \lfloor q \rfloor &= \\
&\{q\} - \{q\}^2 = \{q\}(1 - \{q\}) \geq 0
\end{aligned}$$

By the symmetry, the total number of attacks mounted by any placement of qN queens is at most three times the maximum number of horizontal attacks. Using equation 5.3,

$$Att \leq 3H \leq \frac{3(2D - q + 1)qN}{2} \tag{5.4}$$

Hence, combining equations 5.1 and 5.4 we get

$$qN(2D+1) \leq \frac{3(2D - q + 1)qN}{2} \tag{5.5}$$

Dividing both sides of equation 5.5 by qN and rearranging concludes the proof of the lemma:

$$q \leq \frac{2D+1}{3} \Longrightarrow Q(D,N) \leq \frac{N(2D+1)}{3} \tag{5.6}$$

Q.E.D.

For $N = 1$ our *multi-queen placement problem on a triangle* is equivalent to the problem considered in Nivasch and Lev [54]. Since the number of queens must be integer, we have

$$Q(D,1) \leq \lfloor (2D+1)/3 \rfloor \tag{5.7}$$

By applying Lemma 3 and Lemma 4 we have

Theorem 1 *The coloring produced by the algorithm is legal and the number* $\overline{c}$ *of colors used by the algorithm is bounded by*

$$\frac{N(2D+1)}{3} + N = \frac{N(2D+4)}{3} \tag{5.8}$$

Proof: We are left to prove that no two intersecting dotted intervals (sharing a dot) are colored in the same color. This follows from the following two observations:

- At all times during execution of the algorithm no two overlapping dotted intervals of different periods can have the same color. This is maintained independently of whether the intervals intersect or not.
- In case when the algorithm colors a d-periodic interval starting at i with a color used for another d-periodic interval, these intervals do not intersect at i. Hence they do not intersect at all.

Q.E.D.

Finally, as a corollary of Theorem 1 we have the concluding

Corollary 1 *The approximation ratio of the algorithm is*

$$\frac{2D+4}{3} \tag{5.9}$$

Proof: The optimal coloring uses at least N colors as it is the lower bound on the size of the maximum clique.

Q.E.D.

5.6 Summary

We have shown a simple greedy approximation algorithm for coloring of dotted interval graphs which has better approximation ratio than the algorithms of Aumann et al. [3] and Jiang [37]. Combinatorial lemmas 4 and 5 are generalizations of Nivasch and Lev [54].

Bibliography

[1] Next generation sequencing has lower sequence coverage and poorer SNP-detection capability in the regulatory regions : Scientific reports : Nature publishing group.

[2] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988.

[3] Yonatan Aumann, Moshe Lewenstein, Oren Melamud, Ron Y. Pinter, and Zohar Yakhini. Dotted interval graphs and high throughput genotyping. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 339–348, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[4] Ricardo Baeza-yates, Ro Salinger, and Santiago Chile. Experimental analysis of a fast intersection algorithm for sorted sequences. In *In Proceedings of 12th International Conference on String Processing and Information Retrieval (SPIRE*, pages 13–24. Springer, 2005.

[5] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In *In Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 295–304. ACM Press.

[6] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.

[7] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, December 1976.

[8] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 859–860, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[9] Michael Burrows, M. Burrows, David Wheeler, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[10] Ayelet Butman, Danny Hermelin, Moshe Lewenstein, and Dror Rawitz. Optimization problems in multiple-interval graphs. *ACM Trans. Algorithms*, 6(2):40:1–40:18, April 2010.

[11] Andrea Califano and Isidore Rigoutsos. Flash: A fast look-up algorithm for string homology. In *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology*, pages 56–64. AAAI Press, 1993.

[12] Xin Chen, Ming Li, Bin Ma, and John Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

[13] R. Chenna, H. Sugawara, T. Koike, R. Lopez, T. J. Gibson, D. G. Higgins, and J. D. Thompson. Multiple sequence alignment with the clustal series of programs. *Nucleic Acids Res*, 31(13):3497–3500, July 2003.

[14] Scott Christley, Yiming Lu, Chen Li, and Xiaohui Xie. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, January 2009.

[15] Maxime Crochemore, Gad M. Landau, and Michal Z. Ukelson. A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003.

[16] Matei David, Misko Dzamba, Dan Lister, Lucian Ilie, and Michael Brudno. SHRiMP2: Sensitive yet Practical Short Read Mapping. *Bioinformatics*, 27(7):1011–1012, April 2011.

[17] R. Dementiev and L. Kettner. Stxxl: Standard template library for XXL data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

[18] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, pages 195–208, 2004.

[19] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming I: linear cost functions. *J. ACM*, 39(3):519–545, July 1992.

[20] J.H. Jettand et al. High-speed DNA sequencing: an approach based upon fluorescence detection of single molecules. *Journal of biomolecular structure and dynamics*, 7(2):301–309, 1989.

[21] Timothy D. Harris et al. Single-molecule DNA sequencing of a viral genome. *Science*, 320(5872):106–109, April 2008.

[22] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

[23] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed Text Indexes: From Theory to Practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009. 30 pages.

[24] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *CoRR*, abs/0909.4341, 2009.

[25] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.

[26] Robert W. Floyd. Permuting information in idealized two-level storage. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 105–109. Plenum Press, New York, 1972.

[27] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.

[28] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[29] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[30] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35:378–407, August 2005.

[31] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Data Compression Conference*, 0:213–222, 2006.

[32] J. Hein. A new method that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when the phylogeny is given. *Molecular Biology and Evolution*, 6(6):649–668, 1989.

[33] Danny Hermelin, Julián Mestre, and Dror Rawitz. Optimization problems in dotted interval graphs. In *WG*, pages 46–56, 2012.

[34] Nils Homer, Barry Merriman, and Stanley F. Nelson. BFAST: An Alignment Tool for Large Scale Genome Resequencing. *PLoS ONE*, 4(11):e7767+, November 2009.

[35] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 21(5):734–740, May 2011.

[36] Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Pittsburgh, PA, USA, 1988.

[37] Minghui Jiang. Approximating minimum coloring and maximum independent set in dotted interval graphs. *Inf. Process. Lett.*, 98(1):29–33, April 2006.

[38] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, August 2009.

[39] Christos Kozanitis, Chris Saunders, Semyon Kruglyak, Vineet Bafna, and George Varghese. Compressing genomic sequence fragments using SlimGene. In *RECOMB*, pages 310–324, 2010.

[40] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25+, 2009.

[41] Heng Li. Exploring single-sample snp and indel calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, July 2012.

[42] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, May 2013.

[43] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009.

[44] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, September 1994.

[45] Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002.

[46] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *RECOMB*, pages 121–137, 2009.

[47] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[48] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. Rapid parallel genome indexing with mapreduce. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 51–58, New York, NY, USA, 2011. ACM.

[49] John C. Mu, Hui Jiang, Amirhossein Kiani, Marghoob Mohiyuddin, Narges B. Asadi, and Wing H. Wong. Fast and Accurate Read Alignment for Resequencing. *Bioinformatics*, 28(18):bts450–2373, July 2012.

[50] Gene Myers and Webb Miller. Chaining multiple-alignment fragments in sub-quadratic time. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 38–47, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[51] D. Naor and D.L. Brutlag. On near-optimal alignments of biological sequences. *Journal of Computational Biology*, 1(4):349–266, 1994.

[52] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *JMB*, 48:443–453, 1970.

[53] Zemin Ning, Anthony J. Cox, and James C. Mullikin. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Research*, 11(10):1725–1729, October 2001.

[54] G. Nivasch and E. Lev. *Nonattacking Queens on a Triangle*. December 2005.

[55] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3, November 2007.

[56] Kim R. Rasmussen, Jens Stoye, and Eugene W. Myers. Efficient q-gram filters for finding all epsilon-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, 2006.

[57] T. Rognes and E. Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[58] Torbjrn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12:221, 2011.

[59] Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: Accurate mapping of short color-space reads. *PLoS Comput Biol*, 5(5):e1000386+, May 2009.

[60] Benno Schwikowski and Martin Vingron. Weighted sequence graphs: boosting iterated dynamic programming using locally suboptimal solutions. *Discrete Appl. Math.*, 127(1):95–117, 2003.

[61] SiSoftware. Benchmarks : Intel mobile haswell (CrystalWell): Memory sub-system, 2013. [Online; accessed 22-January-2014].

[62] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[63] Daniel D Sommer, Arthur L Delcher, Steven L Salzberg, and Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC Bioinformatics*, 8:64, February 2007.

[64] M. Waterman T. Smith. Identification of common molecular subsequences. *Journal of molecular biology*, 147:195–197, 1981.

[65] Peter Turnpenny and Sian Ellard. *Emery's Elements of Medical Genetics*. Elsevier, 12 edition, 2004.

[66] Akshat Verma and Sandeep Sen. Combating i-o bottleneck using prefetching: model, algorithms, and ramifications. *The Journal of Supercomputing*, 45(2):205–235, 2008.

[67] W Timothy White and Michael Hendy. Compressing DNA sequence databases with coil. *BMC Bioinformatics*, 9(1):242, 2008.

[68] Wikipedia. XOR Swap Algorithm, 2013. [Online; accessed 10-January-2014].

[69] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, 1997.

[70] Vladimir Yanovsky. Approximation algorithm for coloring of dotted interval graphs. *Inf. Process. Lett.*, 108(1):41–44, September 2008.

[71] Vladimir Yanovsky. ReCoil - an algorithm for compression of extremely large datasets of DNA data. *Algorithms for Molecular Biology*, 6:23, 2011.

[72] Vladimir Yanovsky, Stephen M. Rumble, and Michael Brudno. Read mapping algorithms for single molecule sequencing data. In Keith A. Crandall and Jens Lagergren, editors, *WABI*, volume 5251 of *Lecture Notes in Computer Science*, pages 38–49. Springer, 2008.

[73] Matei Zaharia, William J. Bolosky, Kristal Curtis, Armando Fox, David A. Patterson, Scott Shenker, Ion Stoica, Richard M. Karp, and Taylor Sittler. Faster and more accurate sequence alignment with snap. *CoRR*, abs/1111.5572, 2011.

[74] L. Zane, L. Bargelloni, and T. Patarnello. Strategies for microsatellite isolation: a review. *Molecular ecology*, 11(1):1–16, January 2002.

[75] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang, and Bairong Shen. A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PLoS ONE*, 6(3):e17915, 03 2011.